

COMPANION VOLUME • STAGE 10 OF THE ENGINEER'S SYLLABUS

The Engineer's Shipyard

The previous volume taught you to write code a team can change. This one is about assembling it into a product that strangers use and pay for: the API, the sign-in, the interface, the work behind each request, the money, and the discipline of shipping it and learning from what happens next.

CHAPTERS

20

CODE EXAMPLES

69

EXERCISES

60

STAGE BUDGET

240 h

Contents

How to use this volume	4
Part I · The API	5
Chapter 01 · Designing HTTP APIs	6
Chapter 02 · Pagination, filtering and safe retries	13
Chapter 03 · Evolving an API without breaking clients	19
Chapter 04 · REST, GraphQL, gRPC and real time	26
Part II · Identity and access	32
Chapter 05 · Authentication	33
Chapter 06 · OAuth 2.0 and OpenID Connect	40
Chapter 07 · Authorisation and multi-tenancy	47
Part III · The frontend	54
Chapter 08 · The web platform	55
Chapter 09 · Accessibility	61
Chapter 10 · Components, state and data fetching	68
Chapter 11 · Rendering strategies and web performance	75
Part IV · Behind the request	82
Chapter 12 · Background jobs and scheduled work	83
Chapter 13 · Files, uploads and CDNs	89
Chapter 14 · Webhooks, integrations and email	95
Chapter 15 · Payments	102
Part V · Protecting and releasing	108
Chapter 16 · Abuse, rate limits and input validation	109
Chapter 17 · Releasing safely	116
Part VI · Product and launch	122
Chapter 18 · Product thinking	123
Chapter 19 · Measuring what users do	129

Chapter 20 · Launch and the first ninety days	135
The twenty rules	142
Reading list	144

Each chapter has the same shape as in *The Engineer's Craft*. It opens with its idea in one sentence, works through examples, lists the traps, and ends with exercises, a checklist and further reading. Every exercise has a solution folded underneath it. Try first, then open it.

Build one product while you read. Stage 10's deliverable is a real product with real users, maintained for at least three months. Wherever an exercise can be done against your own product instead of the example, do it there. The chapters are ordered roughly as you'll need them, but Part VI is worth reading early, before you decide what to build.

The running example is a small invoicing app for freelancers and studios. Teams send invoices to their clients, clients pay by card, and the app chases late payers. It needs almost everything in this volume: accounts and teams, an API and a web interface, PDFs and email, payments and webhooks, and a launch.

Code is in Python on the server, TypeScript and React in the browser, SQL for PostgreSQL, and raw HTTP where the protocol itself is the point. The self-contained examples were run before publication to confirm they behave as described. Every idea transfers to other stacks.

THE ONE QUESTION

Tutorials stop at the moment code works on the author's machine. Products begin at the moment it meets the world.

A client retries a payment request because the network blinked. A user pastes an emoji into a name field. Someone navigates by screen reader, or on a mid-range phone on a slow train. A bank asks the cardholder to confirm a payment. An email lands in spam. And someone, within days of launch, deliberately tries to break it.

Every chapter here is about one of those meetings. Before you ship anything, ask **what happens when this meets a real user?** Then ask what happens when it meets an attacker.

PART I

The API

An API is a promise to everyone who calls it, and it's the hardest part of a product to change later. These four chapters cover designing that promise, making it safe to page through and retry, evolving it without breaking anyone, and choosing its shape.

Designing HTTP APIs

3 exercises

A good API is predictable: resources named as nouns, methods that mean what HTTP says they mean, status codes a client can act on, and errors that tell the caller exactly what to fix.

Resources, not actions

Model the nouns of your domain as resources with stable URLs, and let HTTP's methods be the verbs. The URL says *what*; the method says *what to do with it*.

BEFORE remote procedure calls wearing HTTP

```
POST /createInvoice
GET /getInvoice?id=7
POST /updateInvoice
POST /invoice/delete?id=7
GET /getAllInvoicesForClient?clientId=12
```

AFTER resources and methods

```
POST /invoices
GET /invoices/7
PATCH /invoices/7
DELETE /invoices/7
GET /invoices?client_id=cl_12
```

Some operations don't fit create, read, update and delete. Sending an invoice is an action with consequences: an email, a status change, an audit record. Don't contort it into `PATCH /invoices/7 {"status": "sent"}`, which hides the side effects behind what looks like a field edit. `POST /invoices/7/send` is honest, widely understood, and easy to authorise separately. Google's API guidelines call these custom methods. Use them sparingly, and only for genuine actions.

Conventions worth adopting from day one: plural nouns, lower case, hyphens or underscores used consistently, nesting no deeper than one level, and identifiers that are opaque strings such as `inv_8f3k2` rather than sequential integers, which leak your volume and invite people to guess neighbours.

Methods mean what HTTP says they mean

METHOD	MEANING	SAFE	IDEMPOTENT	TYPICAL SUCCESS
GET	Read a resource or collection	Yes	Yes	200
POST	Create a resource, or perform an action	No	No	201 with a <code>Location</code> header, or 200 or 202
PUT	Replace a resource entirely	No	Yes	200 or 204
PATCH	Change part of a resource	No	Not necessarily	200
DELETE	Remove a resource	No	Yes	204

Safe means the request has no side effects the client asked for. **Idempotent** means sending it twice has the same effect as sending it once. RFC 9110 defines both, and browsers, proxies, crawlers and retry libraries all rely on them. A `GET` link that deletes something will eventually be followed by a link prefetcher or a search engine, and your data will quietly disappear.

Status codes a client can act on

The most useful property of a status code is that a client can decide what to do from it alone. A 4xx says "your request is wrong; sending it again unchanged won't help". A 5xx, or a 429, says "try again later".

CODE	MEANING	USE IT WHEN
200 OK	Success, with a body	Reads and most updates
201 Created	A resource was created	Include <code>Location</code> and the new representation
202 Accepted	Work has started, not finished	Exports, PDF generation; point to a status resource
204 No Content	Success, no body	Deletes
400 Bad Request	The request is malformed	Broken JSON, wrong types
401 Unauthorized	Not authenticated, despite the name	Missing or expired credentials
403 Forbidden	Authenticated but not allowed	A viewer tries to delete
	No such resource	Also for resources that exist but that the

404 Not Found

caller mustn't know about

409 Conflict	Conflicts with current state	Deleting an invoice that's already paid
---------------------	------------------------------	---

412 Precondition Failed	A conditional request's condition was false	Stale <code>If-Match</code> version (chapter 2)
--------------------------------	---	---

422 Unprocessable Content	Well-formed but invalid	Due date before issue date
----------------------------------	-------------------------	----------------------------

429 Too Many Requests	Rate limited	Include <code>Retry-After</code> (chapter 16)
------------------------------	--------------	---

500, 502, 503, 504	The server failed	Never for the caller's mistakes
---------------------------	-------------------	---------------------------------

Errors that explain themselves

RFC 9457, *Problem Details for HTTP APIs*, which replaced RFC 7807 in 2023, defines a standard error format with the media type `application/problem+json`. Use it rather than inventing your own. A machine-readable `type` lets code branch on the error; `title` and `detail` are for people; and you can add fields, such as a list of every invalid field.

EXAMPLE one response reports every problem

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

{
  "type": "https://api.example.com/problems/validation",
  "title": "The invoice has invalid fields.",
  "status": 422,
  "errors": [
    {"field": "due_date", "detail": "Must be on or after issue_date (2026-09-01)."},
    {"field": "lines[2].quantity", "detail": "Must be at least 1."}
  ]
}
```

Report all the problems in one response, not one per round trip. Never return a stack trace or a database error message, which leaks internals to anyone probing your API. And never return `200 OK` with an error in the body: every retry library, monitoring tool and cache will believe it succeeded.

Consistency is a feature

Pick conventions once and apply them everywhere. Clients learn one endpoint and can guess the next.

- **Field names:** one case style, such as `snake_case`, across every endpoint.
- **Timestamps:** RFC 3339 strings in UTC, such as `"2026-09-18T14:30:00Z"`. Dates without times as `"2026-09-18"`.
- **Money:** an integer in minor units with its currency, or a decimal string. Never a floating-point number (chapter 15).
- **Enums:** lower-case strings, such as `"overdue"`, never magic integers.
- **Nulls:** decide whether an empty value is `null` or absent, and document it.

EXAMPLE an invoice representation

```
{
  "id": "inv_8f3k2",
  "number": "INV-0042",
  "status": "sent",
  "client": {"id": "cl_q81m", "name": "Studio North"},
  "currency": "EUR",
  "amount_due": {"amount_minor": 125000, "currency": "EUR"},
  "issue_date": "2026-09-01",
  "due_date": "2026-09-30",
  "created_at": "2026-09-01T09:12:44Z",
  "links": {"pdf": "/invoices/inv_8f3k2/pdf", "payments": "/invoices/inv_8f3k2/payments"}
}
```

Write your conventions down in a short API style guide, describe the API in an OpenAPI file, and lint it in CI with a tool such as Spectral. Consistency that isn't checked by a machine erodes one endpoint at a time.

Design from the client's side

Before building an endpoint, write the client code that will call it: the screen that lists invoices, or the integration that syncs them. This is test-first development applied to API design. It exposes chatty designs that need five calls to draw one screen, and it stops you publishing your database schema as your API. Tables are an implementation detail you'll want to change; the API is a contract you'll have to keep.

COMMON TRAPS

Verbs in every URL. `/getAllInvoices` is a function call wearing a URL.

200 with an error body. Every tool between you and the client will think it worked.

Leaking the database. Internal column names, auto-increment IDs and table structure, all

now frozen into a public contract.

Inconsistent shapes. The same invoice returned with different fields by different endpoints.

401 and 403 confused. 401 means "who are you?"; 403 means "I know who you are, and no".

EXERCISES

EXERCISE 1.1 Redesign the endpoints

Redesign these endpoints for the invoicing app as resources and methods.

```
POST /createClient
GET  /getClientInvoices?clientId=12
POST /updateInvoiceDueDate
POST /deleteInvoice?id=7
POST /markInvoicePaid
GET  /searchInvoices?status=overdue
```

SOLUTION

```
POST  /clients
GET   /invoices?client_id=cl_12      (or GET /clients/cl_12/invoices)
PATCH /invoices/7                  {"due_date": "2026-10-01"}
DELETE /invoices/7
POST   /invoices/7/payments          {"amount_minor": 125000, "method":
  "bank_transfer"}
GET    /invoices?status=overdue
```

The interesting one is "mark paid". Model what actually happened, a payment was received, as a resource. The invoice's status then follows from its payments, partial payments work naturally, and you get an audit trail for free. Setting a status field directly would lose all three.

EXERCISE 1.2 Choose the status code

Pick the response code for each situation.

- An invoice is created successfully.
- The request body isn't valid JSON.
- The JSON is valid, but the due date is before the issue date.
- A logged-in user requests an invoice that belongs to another team.
- A user tries to delete an invoice that has already been paid.

- f. A PDF export is requested and will take about thirty seconds.
- g. A client has sent 400 requests in the last minute against a limit of 100.
- h. A request arrives with no credentials at all.

SOLUTION

- a. **201 Created**, with `Location: /invoices/{id}` and the new invoice in the body.
- b. **400 Bad Request**.
- c. **422 Unprocessable Content**, with problem details naming the field.
- d. **404 Not Found**. Don't confirm that the invoice exists. A 403 is defensible only where existence isn't sensitive.
- e. **409 Conflict**, explaining that paid invoices can be refunded but not deleted.
- f. **202 Accepted**, with a `Location` pointing at a status resource such as `/exports/exp_91a` to poll.
- g. **429 Too Many Requests**, with `Retry-After`.
- h. **401 Unauthorized**, with a `WWW-Authenticate` header describing how to authenticate.

EXERCISE 1.3 Write the error

A client submits a new invoice with three problems: the currency is `"usd "` with a trailing space, line 1 has a quantity of `-2`, and `client_id` is missing. Write the complete HTTP response.

SOLUTION

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

{
  "type": "https://api.example.com/problems/validation",
  "title": "The invoice has invalid fields.",
  "status": 422,
  "errors": [
    {"field": "client_id", "detail": "Required."},
    {"field": "currency", "detail": "Must be a three-letter ISO 4217 code such as\n\"USD\"."},
    {"field": "lines[1].quantity", "detail": "Must be at least 1."}
  ]
}
```

There's a design decision hiding in the currency. You could silently trim and upper-case it. That's friendly, but once clients depend on the leniency you can never tighten it again without breaking them. Being strict from day one is the safer default; chapter 3 explains why.

CHECKLIST

- ☐ Are URLs nouns, with HTTP methods as the verbs?
- ☐ Can a client decide whether to retry from the status code alone?
- ☐ Do all errors use one format, report every problem at once, and hide internals?
- ☐ Are naming, timestamps, money and IDs consistent across every endpoint?
- ☐ Did you design from the client's code rather than from the database schema?

GO DEEPER

SPEC	RFC 9110: HTTP Semantics methods, status codes and headers, authoritatively
SPEC	RFC 9457: Problem Details for HTTP APIs
GUIDE	Google API Improvement Proposals a complete, reasoned API style guide
GUIDE	Zalando RESTful API Guidelines
BOOK	The Design of Web APIs Arnaud Lauret

Pagination, filtering and safe retries

3 exercises

Lists must be paged in a way that stays fast and correct while the data changes, and every write must be safe to retry, because on a real network every write eventually will be retried.

Offset pagination, and why it breaks

The obvious design is `?page=3&per_page=50`, which becomes `OFFSET 100 LIMIT 50` in SQL. It's simple, and it lets a user jump straight to page 37. It has two problems that appear only once a product is real.

- **It gets slower the deeper you go.** The database must find and discard every skipped row. Page 1,000 at 50 per page processes 50,000 rows to return 50.
- **It's wrong when data changes.** If a new invoice is created while someone pages through a list sorted newest-first, every row shifts down by one. The last row of page 1 reappears at the top of page 2. Delete a row instead, and one is silently skipped.

Cursor pagination

Instead of "skip N rows", say "continue after this row". This is **keyset pagination**: the query filters on the sort key of the last row the client saw, so the database seeks straight to it through an index. The sort order must be *total*, meaning no two rows can tie, so always add a unique column, usually the id, as a tiebreaker.

Hide the key inside an opaque **cursor** string. If clients could read the cursor's structure, they would start depending on it, and you could never change it.

EXAMPLE `invoices.py` · keyset pagination behind an opaque cursor

```
import base64
import json
from datetime import datetime

MAX_LIMIT = 100

def encode_cursor(created_at: datetime, invoice_id: str) -> str:
    raw = json.dumps({"c": created_at.isoformat(), "i": invoice_id}).encode()
    return base64.urlsafe_b64encode(raw).decode().rstrip("=")

def decode_cursor(cursor: str) -> tuple[datetime, str]:
    padded = cursor + "=" * (-len(cursor) % 4)
    data = json.loads(base64.urlsafe_b64decode(padded))
```

```

return datetime.fromisoformat(data["c"]), data["i"]

def list_invoices(db, team_id: str, cursor: str | None = None, limit: int = 50):
    limit = max(1, min(limit, MAX_LIMIT))
    params = {"team_id": team_id, "limit": limit + 1}    # one extra row: is there more?
    where = "team_id = %(team_id)s"
    if cursor:
        params["c"], params["i"] = decode_cursor(cursor) # a malformed cursor becomes a 400
        where += " AND (created_at, id) < (%(c)s, %(i)s)"
    # Only fixed fragments are joined into the SQL; every value is a bound parameter.
    rows = db.fetch_all(
        f"SELECT * FROM invoices WHERE {where} "
        "ORDER BY created_at DESC, id DESC LIMIT %(limit)s", params)
    page, has_more = rows[:limit], len(rows) > limit
    next_cursor = encode_cursor(page[-1].created_at, page[-1].id) if has_more else None
    return {"data": page, "has_more": has_more, "next_cursor": next_cursor}

```

The query is fast only with a matching index, here on `(team_id, created_at DESC, id DESC)`. The row comparison `(created_at, id) < (...)` compares both columns in order, which is exactly what a total sort needs. The trade-off is that users can't jump to page 37. For feeds, infinite scroll, exports and sync, nobody wants to. For a small admin table where they do, offset pagination is fine.

Filtering and sorting

Accept filters and sort orders as query parameters, such as `?status=overdue&client_id=cl_12&sort=-due_date`, and **allowlist** both. Passing a client-chosen column name into `ORDER BY` invites SQL injection, and even when it's safe, it lets a client pick a sort with no index and run a full table scan on every request. Document the default order, cap the page size, and return 400 for unknown parameters rather than silently ignoring them.

Why every write needs to be retryable

A client sends `POST /invoices/inv_8f3k2/payments`. The server charges the card. The response is lost on the way back, because a phone switched networks. From the client's side, the request simply timed out. Did it work? It can't know. Retrying might charge the card twice; not retrying might lose the payment.

The standard answer is an **idempotency key**. The client generates a unique key for each logical operation, once, when the user clicks "Pay", and sends it with every attempt in an `Idempotency-Key` header. The server stores the key with a fingerprint of the request and the response it produced. A repeated key gets the stored response back instead of a second charge. Stripe popularised the pattern, an IETF draft describes the header, and Stripe discards keys after at least 24 hours, comfortably longer than any sensible retry window.

EXAMPLE `idempotency.py`

```

import hashlib
import json

```

```
def with_idempotency(db, key: str, request_body: dict, handler):
    fingerprint = hashlib.sha256(
        json.dumps(request_body, sort_keys=True, separators=(",", ":")).encode()
    ).hexdigest()
    try:
        db.execute(
            "INSERT INTO idempotency_keys (key, fingerprint, status) "
            "VALUES (%s, %s, 'started'", (key, fingerprint))
    except UniqueViolation:
        record = db.fetch_one(
            "SELECT fingerprint, status, response_code, response_body "
            "FROM idempotency_keys WHERE key = %s", (key,)
        )
        if record.fingerprint != fingerprint:
            return 422, problem("idempotency-key-reused",
                                "This key was already used for a different request.")
        if record.status == "started":
            return 409, problem("request-in-progress",
                                "The original request is still being processed.")
        return record.response_code, json.loads(record.response_body) # replay

    code, body = handler()
    db.execute(
        "UPDATE idempotency_keys SET status = 'done', response_code = %s, "
        "response_body = %s WHERE key = %s", (code, json.dumps(body), key))
    return code, body
```

Two details matter in production. Run the handler's writes and the final update in one transaction, and delete the `started` row if the handler fails, so a retry can try again. And scope keys to the caller, so one customer's key can never collide with another's.

Concurrent edits: lost updates and ETags

Two people open the same invoice. Both edit the notes and save. The second save silently overwrites the first, and nobody is told. This is a **lost update**, and "last write wins" is its polite name. HTTP's fix is optimistic concurrency with `ETag` and `If-Match`.

EXAMPLE the second save fails loudly

```
GET /invoices/inv_8f3k2
→ 200 OK
   ETag: "v7"

PATCH /invoices/inv_8f3k2
If-Match: "v7"
Content-Type: application/merge-patch+json

{"notes": "Net 30. Thanks for your business."}

→ 412 Precondition Failed      (someone saved version 8 in the meantime)
```

On the server, a version column does the work: `UPDATE invoices SET notes = %s, version = version + 1 WHERE id = %s AND version = %s`. If no row was updated, the version moved on, so return 412. The client can then show "this invoice changed since you opened

it" and offer to reload, instead of destroying a colleague's work.

What PATCH means

JSON Merge Patch (RFC 7396) is the common choice: send only the fields to change, and `null` means "clear this field". It's simple, but it can't say "remove the third line item". JSON Patch (RFC 6902) sends a list of operations and can, at the cost of complexity. Whichever you choose, state it with the request's media type, and document the difference between a field that's absent (leave it alone) and one that's null (clear it).

COMMON TRAPS

Unbounded lists. An endpoint with no maximum page size will one day return two million rows.

Sorting on a column that ties. Rows with equal timestamps come back in any order, so pages skip and repeat.

Readable cursors. Clients start constructing them, and you can never change the format.

Retrying POST without a key. Duplicate invoices, double charges, repeated emails.

Keys without fingerprints. A reused key with a different body silently returns the wrong response.

Last write wins on shared data. Somebody's work disappears, and nobody finds out.

EXERCISES

EXERCISE 2.1 Spot the pagination bugs

This query backs an infinite-scroll list of invoices, and new invoices are created all day. List every problem, then fix it.

```
SELECT * FROM invoices
WHERE team_id = $1
ORDER BY due_date
LIMIT 50 OFFSET $2;
```

SOLUTION

- **The order isn't total.** Many invoices share a due date, so rows with equal dates can come back in a different order on each request, and pages overlap or skip.
- **Offsets shift** as invoices are created or deleted, causing duplicates and gaps.
- **Deep offsets are slow**, because every skipped row is still read.
- **There may be no index** supporting this order within a team.

```
-- Index: CREATE INDEX ON invoices (team_id, due_date, id);
SELECT * FROM invoices
WHERE team_id = $1
      AND (due_date, id) > ($2, $3)      -- the cursor: last row's due date and id
ORDER BY due_date, id
LIMIT 51;                               -- one extra row tells you whether there's more
```

EXERCISE 2.2 Trace the retries

Using `with_idempotency` above, what does the client get back in each case?

- It retries with the same key after the first request has completed.
- It retries with the same key while the first request is still running.
- It retries with the same key but a different amount.
- It retries with a newly generated key.

SOLUTION

- The stored response is replayed: the same 201 and the same payment id. The card is charged once.
- 409, "still being processed". The client should wait and retry with the same key.
- 422, "key reused for a different request". This is almost always a client bug, and failing loudly is correct.
- The server sees a brand-new operation and **charges again**. That's why the key must be generated once per user intention, when the button is clicked, and reused for every attempt, never regenerated per HTTP request.

EXERCISE 2.3 Prevent the lost update

Two browser tabs are editing the same invoice's notes. Write the SQL for the save and describe what the second tab sees when it saves.

SOLUTION

```
UPDATE invoices
SET notes = $1, version = version + 1, updated_at = now()
WHERE id = $2 AND version = $3;    -- $3 is the version this tab loaded
```

The first save updates one row, and the version becomes 8. The second save matches no rows, because the version is no longer 7. The server returns 412 Precondition Failed, and the second tab shows "This invoice was changed by someone else", with the other version and an option to reload or copy their text. Nobody's work vanishes silently.

CHECKLIST

- ☐ Is every list paginated, with a maximum page size?
- ☐ Is every sort order total, with a unique tiebreaker and a matching index?
- ☐ Can every non-idempotent write be retried safely with an idempotency key?
- ☐ Do concurrent edits fail loudly instead of silently overwriting?
- ☐ Are filter and sort fields allowlisted?

GO DEEPER

ESSAY No Offset Markus Winand, on keyset pagination

DOCS Idempotent requests Stripe's implementation

ESSAY Implementing Stripe-like Idempotency Keys in Postgres Brandur Leach

DRAFT The Idempotency-Key HTTP Header Field IETF

Evolving an API without breaking clients

3 exercises

Once someone depends on your API, every change is a negotiation. Most changes can be made additively. For the rest, version deliberately, deprecate loudly, and give people time.

What counts as breaking

A change is breaking if a client that worked yesterday stops working today without changing anything. That's a broader category than most people assume.

CHANGE	BREAKING?	WHY
Add an optional request field	No	Old clients don't send it
Add a new endpoint	No	Nobody calls it yet
Add a field to a response	Usually not	Tolerant clients ignore it; strictly generated clients may not
Add a value to a response enum	Often	A client's <code>switch</code> meets <code>"disputed"</code> and falls over
Remove or rename a field	Yes	Code reading it gets nothing
Make an optional field required	Yes	Existing requests start failing
Change a type or format	Yes	Cents to decimal, integer to string
Tighten validation	Yes	Requests that were valid now return 422
Change a default order, page size or rate limit	In practice, yes	Somebody relied on it

That last row is Hyrum's law, from *The Engineer's Craft*: with enough users, every observable behaviour is depended on by somebody, whatever your documentation promises. Assume undocumented behaviour is load-bearing until you've checked.

Be strict early, tolerant as a client

Postel's robustness principle says to be conservative in what you send and liberal in what you accept. RFC 9413, published in 2023, argues that liberal acceptance entrenches bugs: once you accept a malformed input, clients depend on it, and you can never stop. For API design the lesson splits in two.

- **As a server, validate strictly from day one.** Loosening validation later is easy and never breaks anyone. Tightening it always breaks someone.
- **As a client, read tolerantly.** Ignore fields you don't recognise, and handle enum values you've never seen with a sensible default. Martin Fowler calls this the tolerant reader. Your own frontend and SDKs should be tolerant readers of your API.

Versioning strategies

STRATEGY	EXAMPLE	STRENGTHS	COSTS
URL path	/v2/invoices	Obvious, cacheable, easy to route and test	Every break bumps the whole API; clients must change URLs
Media type or header	Accept: application/vnd.example.v2+json	Clean URLs, per-resource versions	Invisible in logs and browsers; easy to get wrong
Date-pinned versions	Stripe-Version: 2024-06-20	Small, frequent changes; each customer upgrades on their own schedule	The server must translate between every supported version

Stripe's approach is the benchmark for public APIs. Each account is pinned to the API version current when it first made a request, and the server keeps a chain of small transformations that turn today's responses into each older shape. It's elegant and it costs real engineering. For most products, a better default is to evolve additively almost always, and reserve a new URL version for the rare, genuinely incompatible redesign.

Expand and contract

Most "breaking" changes can be made as a series of non-breaking ones. Suppose you want to replace an invoice's flat `client_name` field with a nested `client` object.

- 01 **Expand.** Add `client` alongside `client_name`, and return both. Nothing breaks.
- 02 **Migrate.** Update your documentation, examples and SDKs to use the new field. Mark the old one as deprecated in the OpenAPI description.

- 03 **Measure.** Find out who still uses the old field. For request fields, you can log it per API key. For response fields you usually can't see reads, so rely on SDK versions, surveys and direct contact.
- 04 **Announce** a sunset date, with deprecation headers and emails to the affected developers.
- 05 **Contract.** Remove the old field in the next major version, after the date has passed.

It's the same pattern as the database migrations of chapter 17, and as branch by abstraction in *The Engineer's Craft*: never change a contract in one step when you can change it in several reversible ones.

Deprecating out loud

Two standard response headers tell clients about deprecation. RFC 9745, published in 2025, defines `Deprecation`, whose value is the moment the endpoint became deprecated. RFC 8594 defines `Sunset`, the date it will stop working, which must not be earlier than the deprecation date. A `Link` header with `rel="deprecation"` points at the migration guide.

EXAMPLE deprecated from 1 October 2026, removed on 1 April 2027

```
HTTP/1.1 200 OK
Content-Type: application/json
Deprecation: @1790812800
Sunset: Thu, 01 Apr 2027 00:00:00 GMT
Link: <https://api.example.com/docs/migrate-to-v2>; rel="deprecation"
```

Headers alone aren't enough, because few clients read them. Email the developers behind every API key that still calls the old endpoint, which is one reason every key should be tied to a real contact. Put the change in your changelog. Before the sunset date, run scheduled **brownouts**: switch the old endpoint off for an announced hour or two, so hidden dependencies break while there's still time to fix them. After the sunset, return `410 Gone` with problem details pointing at the migration guide, not a bare 404.

The contract as a file

Describe the API in an OpenAPI document and treat it as the source of truth. It generates reference documentation, client SDKs and mock servers. More importantly, it can be *diffed*: a CI step comparing the new specification with the last release, using a tool such as `oasdiff`, flags removed fields, new required parameters and narrowed types before they reach a customer.

EXAMPLE openapi.yaml, excerpt

```
components:
  schemas:
```

```

Invoice:
  type: object
  required: [id, number, status, amount_due]
  properties:
    id: {type: string, example: inv_8f3k2}
    number: {type: string, example: INV-0042}
    status:
      type: string
      enum: [draft, sent, paid, overdue, void]
      description: New values may be added. Treat unknown values as "sent".
    client_name:
      type: string
      deprecated: true
      description: Use client.name. Removed on 2027-04-01.
    client: {$ref: "#/components/schemas/ClientSummary"}

```

Note the description on `status`. Telling clients up front that the enum may grow, and how to treat unknown values, turns a future breaking change into a non-event.

COMMON TRAPS

"We only added a field." New enum values and strict deserialisers make additions break things too.

A new version for every change. Version 14 after two years means fourteen APIs to maintain.

Deprecating blind. If you can't name who uses an endpoint, you can't warn them.

Removing on the date, unannounced. Without brownouts, the first sign of trouble is a customer's outage.

Internal models as the API. When the API mirrors your tables, every refactor becomes a breaking change.

EXERCISES

EXERCISE 3.1 Breaking or not?

For each change to the invoicing API, say whether it's breaking, and why.

- Add an optional `notes` field when creating an invoice.
- The `status` field in responses gains a new value, `disputed`.
- `amount` changes from integer cents to a decimal string.
- The default sort of `GET /invoices` changes from creation date to due date.

- e. The rate limit drops from 100 to 60 requests a minute.
- f. The wording of a validation error changes.
- g. A new endpoint, `GET /invoices/{id}/events`, is added.

SOLUTION

- a. **Not breaking.** Existing clients never send it.
- b. **Potentially breaking**, unless you documented that the enum can grow. Announce it before it appears.
- c. **Breaking.** Every client parsing the number fails, or worse, misreads it.
- d. **Breaking in practice.** Somebody takes "the first invoice" as "the newest".
- e. **Breaking** for clients running between 60 and 100 a minute. Announce it, and give heavy users notice.
- f. **Formally not.** But someone may be matching on the text, which is why errors need a stable `type` for code to use.
- g. **Not breaking.**

EXERCISE 3.2 Plan the rename

You want to replace `client_name` with a nested `client: {id, name}` object. List the releases and communications, in order, with rough timings.

SOLUTION

1. **Week 0:** release returning both `client` and `client_name`. Update docs and SDKs to use `client`; mark `client_name` deprecated in the spec.
2. **Week 1:** changelog entry, and email to every developer with an active key, giving a sunset date six months out.
3. **Weeks 1 to 24:** send `Deprecation`, `Sunset` and `Link` headers. Track SDK versions in use and contact heavy users directly.
4. **Weeks 20 and 22:** announced brownouts, removing `client_name` for an hour.
5. **Week 26:** remove the field in the next major version, and keep the migration guide online.

Six months suits a public API. For a partner integration, agree the timeline with them. For your own frontend alone, you can move in days.

EXERCISE 3.3 After the sunset

Write the response a client receives when calling `GET /v1/invoices` after its sunset date has passed.

SOLUTION

```
HTTP/1.1 410 Gone
Content-Type: application/problem+json
Link: <https://api.example.com/docs/migrate-to-v2>; rel="deprecation"

{
  "type": "https://api.example.com/problems/endpoint-retired",
  "title": "This endpoint was retired on 1 April 2027.",
  "status": 410,
  "detail": "Use GET /v2/invoices. The migration guide explains every difference."
}
```

410 tells the client, and anyone debugging it, that this is permanent and deliberate. A 404 would look like a typo or an outage.

CHECKLIST

- ☐ Is every change classified as breaking or not, including undocumented behaviour?
- ☐ Do your own clients tolerate unknown fields and enum values?
- ☐ Can you name every consumer of an endpoint before deprecating it?
- ☐ Do deprecated endpoints send `Deprecation`, `Sunset` and `Link` headers?
- ☐ Does CI diff the API specification against the last release?

GO DEEPER

SPEC RFC 9745: The Deprecation HTTP Response Header Field

SPEC RFC 8594: The Sunset HTTP Header Field

ESSAY APIs as infrastructure: future-proofing Stripe with versioning Brandur Leach

SPEC RFC 4181: Media Type Registration

RFC 9413: Maintaining Robust Protocols the case against being liberal in what you accept

REST, GraphQL, gRPC and real time

3 exercises

Each API style optimises for a different constraint. Choose by who your clients are and how they fetch data, not by fashion. Most products need plain HTTP plus, at most, one real-time channel.

JSON over HTTP

The style of chapters 1 to 3, usually called REST, whether or not it meets every one of Roy Fielding's constraints. Its strengths are that it's universal: every language and tool speaks it, curl can debug it, HTTP caching and CDNs work with it unmodified, and OpenAPI gives it a schema. Its weaknesses show on complex screens. A client may **over-fetch**, receiving fifty fields to show three, or **under-fetch**, needing five requests to draw one dashboard.

GraphQL

Facebook open-sourced GraphQL in 2015. The client sends a query naming exactly the fields it needs, across related objects, to a single endpoint, and a strongly typed schema describes everything available. It shines when many different clients, such as web, iOS, Android and partners, need different slices of the same data, and when a frontend team wants to move without waiting for new endpoints.

EXAMPLE one query for a whole screen

```
query OverdueInvoices {
  invoices(first: 20, status: OVERDUE) {
    edges {
      node {
        number
        dueDate
        amountDue { amountMinor currency }
        client { name email }
      }
    }
    pageInfo { hasNextPage endCursor }
  }
}
```

The costs are real, and they land on the server.

- **N+1 queries.** A naive resolver for `client` runs once per invoice: twenty invoices, twenty-one queries. The fix is a **DataLoader**, which collects every key requested during one tick and fetches them in a single batch.

- **Hostile queries.** A deeply nested query can ask for invoices, their clients, the clients' invoices, and so on, until the server falls over. Limit depth, compute a cost for each query, and consider allowing only pre-registered persisted queries in production.
- **Caching.** Every query is a POST to one URL, so HTTP caches and CDNs can't help without persisted queries.
- **Authorisation per field.** Because a client can reach any field through any path, checks must live in the resolvers, not in a handful of endpoints.
- **Errors.** A GraphQL response is usually `200 OK` with an `errors` array, so monitoring by status code sees nothing.

gRPC

gRPC sends Protocol Buffers over HTTP/2. You define services and messages in `.proto` files, and code generators produce typed clients and servers for many languages. It's compact, fast, supports streaming in both directions, and makes breaking changes visible at compile time. That makes it an excellent choice between your own services. It's a poor choice for public or browser-facing APIs: browsers can't speak it natively and need gRPC-Web or a proxy, and binary payloads can't be read by a human with curl.

EXAMPLE `invoices.proto`

```
syntax = "proto3";

service InvoiceRenderer {
  rpc RenderPdf (RenderPdfRequest) returns (RenderPdfResponse);
}

message RenderPdfRequest {
  string invoice_id = 1;
  string locale = 2;
}

message RenderPdfResponse {
  string storage_key = 1;
  int64 page_count = 2;
}
```

In an all-TypeScript codebase, typed remote procedure call libraries such as tRPC offer similar end-to-end type safety without a schema language, at the cost of tying the client and server tightly together.

Real time: polling, SSE and WebSockets

TECHNIQUE	DIRECTION	GOOD FOR	WATCH OUT FOR
Polling a	Client asks	Rare updates; long jobs (the 202 pattern)	Wasted requests; latency equals the interval

status
resource

from chapter 1)

Server-sent events	Server to client, over plain HTTP	Notifications, progress, live feeds	One-way only; text only; per-domain connection limits on HTTP/1.1
WebSockets	Both ways	Chat, collaborative editing, multiplayer	Long-lived stateful connections complicate load balancing and deploys; you need heartbeats and reconnection

Most features described as "real-time" only need the server to push, and server-sent events do that with ordinary HTTP. The browser's `EventSource` reconnects automatically, and if the server labels each event with an `id`, the browser sends the last one it saw in a `Last-Event-ID` header so the server can resume where it left off.

EXAMPLE listening for payments in the browser

```
const events = new EventSource("/invoices/events");

events.addEventListener("invoice.paid", (event) => {
  const { id, paidAt } = JSON.parse(event.data);
  markInvoicePaid(id, paidAt);
});

events.addEventListener("error", () => {
  // EventSource retries on its own. Show a quiet "reconnecting" state,
  // and refetch the list on reconnect in case an event was missed.
  showConnectionWarning();
});
```

How to choose

- **A public API for other developers:** JSON over HTTP, described in OpenAPI, with webhooks (chapter 14) for events.
- **Your own web and mobile apps, with varied, data-heavy screens and a dedicated frontend team:** GraphQL can pay for itself. So can a backend-for-frontend that exposes screen-shaped endpoints.
- **High-volume calls between your own services:** gRPC.
- **The server needs to push to a browser:** server-sent events first; WebSockets only if the browser must talk back continuously.
- **A new product with one web client:** plain JSON over HTTP. You can add the others when a real need appears. Removing them is much harder.

COMMON TRAPS

GraphQL without limits. No depth or cost limits means any client can take the server down.

WebSockets for one-way updates. All of the operational cost, none of the benefit.

gRPC at the browser edge. You'll spend your time on proxies and tooling.

Real time without recovery. Connections drop; if nothing refetches on reconnect, users see stale data and never know.

Choosing by fashion. The API style your favourite company uses solved their problem, not yours.

EXERCISES

EXERCISE 4.1 Pick the style

Choose an API style or channel for each need, and give one reason.

- Accounting software from other companies syncs invoices out of the app.
- The app's dashboard shows invoices, clients, payments and recent activity on one screen.
- The API service calls the PDF rendering service about 200 times a second.
- A freelancer should see "Studio North viewed your invoice" appear without refreshing.

SOLUTION

- JSON over HTTP with OpenAPI, plus webhooks.** External developers expect it, every language supports it, and webhooks spare them from polling.
- Either one screen-shaped endpoint, or GraphQL** if many such screens and several clients exist. For one web app, a dedicated `GET /dashboard` endpoint is simpler.
- gRPC.** Internal, high volume, and both sides are yours, so typed contracts and binary efficiency pay off.
- Server-sent events.** The server pushes; the browser never needs to send anything back on the channel.

EXERCISE 4.2 Batch the resolver

This GraphQL resolver runs one query per invoice. Rewrite it with a DataLoader so that a page of invoices makes one query for all their clients. Assume the Strawberry library for Python.

```
async def resolve_client(invoice, info):
    return await db.fetch_one("SELECT * FROM clients WHERE id = %s", (invoice.client_id,))
```

SOLUTION

```
from strawberry.dataloader import DataLoader

async def load_clients(ids: list[str]) -> list[Client | None]:
    rows = await db.fetch_all("SELECT * FROM clients WHERE id = ANY(%s)", (ids,))
    by_id = {row.id: row for row in rows}
    return [by_id.get(client_id) for client_id in ids]    # same order as the keys

def get_context():
    return {"client_loader": DataLoader(load_fn=load_clients)}    # one per request

async def resolve_client(invoice, info):
    return await info.context["client_loader"].load(invoice.client_id)
```

Two details matter. The loader must return results in the same order as the keys it was given, including a placeholder for missing ones. And it must be created per request, never shared, or one user's cached data could leak into another user's response.

EXERCISE 4.3 Design an event stream

Design a server-sent event stream that reports progress on a PDF export of a year's invoices. Name the events, their data, and how the stream behaves on reconnection and completion.

SOLUTION

```
id: 14
event: export.progress
data: {"export_id": "exp_91a", "done": 140, "total": 312}

id: 15
event: export.ready
data: {"export_id": "exp_91a", "download_url": "/exports/exp_91a/download"}

: keep-alive          (a comment line every 15 seconds so proxies keep the connection)
```

- Every event carries an increasing `id` . On reconnection, the server reads `Last-Event-ID` and replays anything newer.
- A third event, `export.failed` , carries a reason a person can act on.
- After `ready` or `failed` , the server ends the stream.
- The export also exists as `GET /exports/exp_91a` , so a client that can't hold a connection can simply poll.

CHECKLIST

- ☐ Did you choose the style from your clients' needs rather than from fashion?
- ☐ With GraphQL: depth and cost limits, batching, and per-field authorisation?
- ☐ For real time: the simplest channel that works, with reconnection and replay?
- ☐ Is there a plain status resource for clients that can't hold a connection?

GO DEEPER

DOCS [Learn GraphQL including best practices for pagination and performance](#)

DOCS [Introduction to gRPC](#)

DOCS [Using server-sent events MDN](#)

BOOK [Designing Web APIs Brenda Jin, Saurabh Sahni & Amir Shevat · REST, GraphQL, RPC, webhooks and real time](#)

PART II

Identity and access

Who is this, and what may they do? Getting either question wrong is the most common way real products leak data. Three chapters: proving identity, delegating it with OAuth, and deciding what each identity may touch.

Authentication

3 exercises

Authentication proves who someone is. Store as little as you can, hash what you must, keep sessions in locked-down cookies, and treat account recovery as part of the login, because attackers do.

Build or buy?

For most products, use your framework's built-in authentication or a managed identity provider rather than writing your own. Authentication is full of subtle, security-critical details, and the maintained options have already fixed the bugs you haven't met yet. But you still need everything in this chapter, because you have to configure those tools correctly, judge whether they're any good, and debug them when a user can't log in.

Passwords, if you have them

Never store a password. Store a **hash** made by a deliberately slow, memory-hard algorithm, so that a stolen database is expensive to crack. OWASP's current guidance is Argon2id with at least 19 MiB of memory, 2 iterations and a parallelism of 1. Fall back to scrypt if Argon2id isn't available, and use bcrypt, with a cost of at least 10, only in legacy systems, remembering that it ignores everything after 72 bytes. Fast hashes such as SHA-256 on their own are wrong for passwords, and encryption is wrong too, because it's reversible.

EXAMPLE passwords.py, using the argon2-cffi library

```
from argon2 import PasswordHasher
from argon2.exceptions import VerifyMismatchError

hasher = PasswordHasher(time_cost=2, memory_cost=19_456, parallelism=1)  # memory in KiB

def hash_password(password: str) -> str:
    return hasher.hash(password)  # a random salt is generated and embedded

def verify_password(stored_hash: str, attempt: str) -> bool:
    try:
        return hasher.verify(stored_hash, attempt)
    except VerifyMismatchError:
        return False

def needs_rehash(stored_hash: str) -> bool:
    return hasher.check_needs_rehash(stored_hash)  # upgrade parameters at next login
```

What to demand of the password itself changed in 2025, when NIST finalised SP 800-63B-4.

- **Length over complexity.** At least 15 characters when a password is the only factor, and at least 8 when it's one factor of several.
- **No composition rules.** Stop requiring a symbol, a digit and a capital letter. They produce `Password1!` .
- **Screen against breached and common passwords** instead, for example with the k-anonymity range lookup of Have I Been Pwned, which never sends the password itself.
- **Allow long passwords, spaces and Unicode**, at least 64 characters, and never block pasting. Password managers depend on it.
- **No forced periodic changes.** Require a change only when there's evidence of compromise.

Sessions and cookies

After a successful login, create a session: a long random identifier, at least 128 bits from a cryptographically secure source, stored on the server with the user it belongs to. Send only that identifier to the browser, in a cookie with every protective attribute set.

EXAMPLE a session cookie

```
Set-Cookie: __Host-session=Xk3vQ9...; Path=/; Secure; HttpOnly; SameSite=Lax; Max-Age=1209600
```

- **HttpOnly** stops JavaScript reading the cookie, so a cross-site scripting bug can't simply steal the session.
- **Secure** sends it only over HTTPS.
- **SameSite=Lax** withholds it from cross-site form posts and embedded requests, which blunts cross-site request forgery.
- The **__Host-** prefix makes the browser refuse the cookie unless it's secure, has `Path=/` , and has no `Domain` , so a subdomain can't overwrite it.

Issue a *new* session identifier at login, and again whenever privileges change, which defeats session fixation. On logout, delete the session on the server, not just the cookie. Give sessions an idle timeout and an absolute lifetime, and let users see their active sessions and sign out of all of them.

Sessions or JWTs?

ASPECT	SERVER-SIDE SESSIONS	SELF-CONTAINED TOKENS (JWTs)
Revoking	Delete the row; takes effect immediately	Valid until expiry, unless you add a denylist, which is a session store again

Per request	One lookup in a database or cache	Verify a signature
Size	A short random string	Hundreds of bytes or more
Best for	First-party web apps	Short-lived access tokens between services, or from an identity provider

For a web app talking to its own backend, server-side sessions in `HttpOnly` cookies are simpler and safer. A JWT is a token format, not a session strategy. If you do use JWTs, verify the signature with an algorithm you choose, never the one named in the token, and check issuer, audience and expiry every time. Never keep them in `localStorage`, where any script on the page can read them.

Cross-site request forgery

Browsers attach cookies automatically, so another site can try to make your user's browser submit a form to your app. `SameSite=Lax` blocks cookies on cross-site POSTs but still sends them on ordinary top-level links, which is one more reason a GET must never change anything. Add a second layer for every state-changing request: a CSRF token tied to the session, or a check that the `Origin` or `Sec-Fetch-Site` request header says the request came from your own site.

Multi-factor authentication and passkeys

A second factor stops most account takeovers that start with a leaked password. The options differ a lot in strength.

- **Passkeys** are public-key credentials created by the user's device and bound to your site's domain. There is no shared secret to steal from your database, and a phishing page on a lookalike domain simply can't use them. Synced passkeys are backed up by the platform across the user's devices, and NIST's 2025 guidance recognises them. Offer them as a primary sign-in method, and make them appear in the browser's autofill so people actually find them.
- **Authenticator apps** generate time-based one-time codes. Good, but a convincing phishing page can relay them in real time.
- **SMS codes** are the weakest: SIM-swap fraud and message interception defeat them. Better than nothing, but not a foundation.

Whatever you offer, issue single-use recovery codes when a user enrolls, and store them hashed, like passwords.

Account recovery is part of the login

An attacker doesn't need to defeat your login if your "forgot password" flow is weaker. The rules:

- Reset tokens are long and random, single-use, and expire within about an hour.
- Store only a hash of the token, so a leaked database doesn't hold working reset links. A fast hash is fine here, because the token is random rather than chosen by a person.
- Say the same thing whether or not the email exists: "If an account exists for that address, we've sent a link."
- Rate-limit reset requests, and notify the user by email whenever their password or second factor changes.
- After a reset, end every other session.
- Recovery must not be weaker than sign-in. A passkey protected by an SMS fallback is only as strong as SMS.

EXAMPLE password_resets.py

```
import hashlib
import secrets
from datetime import datetime, timedelta, timezone

RESET_TTL = timedelta(minutes=30)

def create_reset_token(db, user_id: str) -> str:
    token = secrets.token_urlsafe(32)           # 256 bits of randomness
    db.execute(
        "INSERT INTO password_resets (token_hash, user_id, expires_at) VALUES (%s, %s, %s)",
        (hashlib.sha256(token.encode()).hexdigest(), user_id,
         datetime.now(timezone.utc) + RESET_TTL))
    return token                               # sent by email, never stored

def consume_reset_token(db, token: str) -> str | None:
    row = db.fetch_one(
        "DELETE FROM password_resets WHERE token_hash = %s AND expires_at > now() "
        "RETURNING user_id",                    # deleting it makes it single-use
        (hashlib.sha256(token.encode()).hexdigest(),))
    return row.user_id if row else None
```

Throttling and enumeration

Limit login attempts per account and per IP address (chapter 16), with growing delays rather than permanent lockouts, which an attacker could use to lock real users out. Return the same message, taking about the same time, for "no such user" and "wrong password", so attackers can't discover which emails have accounts. Sign-up is the hard case: "this email is already registered" leaks the same fact. One answer is to accept the sign-up form either way and email the address, telling an existing user that someone tried to register with it.

COMMON TRAPS

Home-made crypto. Your own session format, token scheme or hashing. Use maintained libraries.

Tokens in localStorage. Readable by every script on the page, including third-party ones.

Account existence leaks through login, reset and sign-up messages or timings.

Hostile password rules. Blocking paste, demanding symbols, forcing rotation.

Recovery weaker than login. The attacker walks around the front door.

Sessions that survive a password change. The attacker stays logged in.

EXERCISES

EXERCISE 5.1 Audit the cookie

List everything wrong with this login response.

```
Set-Cookie: session=eyJ1c2VyX2lkIjogNDIsICJyb2x1IjogIm1lbWJlciJ9; Path=/; Max-Age=31536000
```

SOLUTION

- **The value is data, not a random identifier.** It's base64 for `{"user_id": 42, "role": "member"}`. Unless it's signed and checked, a user can edit it into `"admin"`, or into someone else's id.
- No `HttpOnly`, so any script can read it.
- No `Secure`, so it can travel over plain HTTP.
- No `SameSite`, leaving cross-site request forgery wide open in older browsers.
- A one-year lifetime with no server-side session to revoke.
- No `__Host-` prefix, so a compromised subdomain could set or overwrite it.

```
Set-Cookie: __Host-session=Xk3vQ9b2...; Path=/; Secure; HttpOnly; SameSite=Lax; Max-Age=1209600
```

EXERCISE 5.2 Review the reset flow

A product's password reset works like this. The token is the base64 encoding of the user's id and the current time. Tokens stay valid for seven days and can be used repeatedly. Unknown emails get "No account with that address". After a reset, the user's other sessions carry on. Find every problem.

SOLUTION

- **Guessable token.** An id and a timestamp can be forged by anyone. Use `secrets.token_urlsafe(32)`.
- **Seven days is far too long.** Thirty to sixty minutes is typical.
- **Reusable.** Delete the token when it's used, and invalidate outstanding tokens when the password changes.
- **Account enumeration.** Respond identically for every address.
- **Surviving sessions.** If an attacker was logged in, they still are. End all other sessions and notify the user by email.
- **Unstated but likely:** the token stored in plain text, and no rate limit on reset requests.

EXERCISE 5.3 Write the login's failure path

Write a login function that throttles attempts, returns one message for every failure, takes similar time whether or not the user exists, and starts a fresh session on success. Use `verify_password` from above.

SOLUTION

```
DUMMY_HASH = hash_password("placeholder-password-for-timing")

def login(db, limiter, email: str, password: str, ip: str):
    email = email.strip().lower()
    if limiter.too_many_failures(email=email, ip=ip):
        raise TooManyAttempts() # 429 with Retry-After
    user = db.find_user_by_email(email)
    stored = user.password_hash if user else DUMMY_HASH
    password_ok = verify_password(stored, password) # always run the slow hash
    if user is None or not password_ok:
        limiter.record_failure(email=email, ip=ip)
        raise InvalidCredentials("Email or password is incorrect.")
    limiter.clear(email=email)
    if needs_rehash(user.password_hash):
        db.update_password_hash(user.id, hash_password(password))
    return start_new_session(user) # a fresh id defeats fixation
```

Verifying against a dummy hash when the user doesn't exist keeps the response time similar in both cases. Without it, a missing account answers in a millisecond and a real one takes a hundred, which tells an attacker exactly which emails are registered.

CHECKLIST

- ☐ Are passwords hashed with Argon2id, or scrypt or bcrypt, and screened against breached lists?
- ☐ Are sessions random, server-side, and sent in `__Host-`, `HttpOnly`, `Secure`, `SameSite` cookies, rotated at login?
- ☐ Do you offer a second factor, ideally passkeys, with recovery codes?
- ☐ Is account recovery at least as strong as login, and silent about which emails exist?
- ☐ Are login and reset requests rate-limited?

GO DEEPER

GUIDE [Authentication Cheat Sheet OWASP](#)

GUIDE [Password Storage Cheat Sheet OWASP](#)

GUIDE [Session Management Cheat Sheet OWASP](#)

SPEC [NIST SP 800-63B-4 authentication guidelines, finalised 2025](#)

GUIDE [passkeys.dev implementing passkeys, for developers](#)

OAuth 2.0 and OpenID Connect

3 exercises

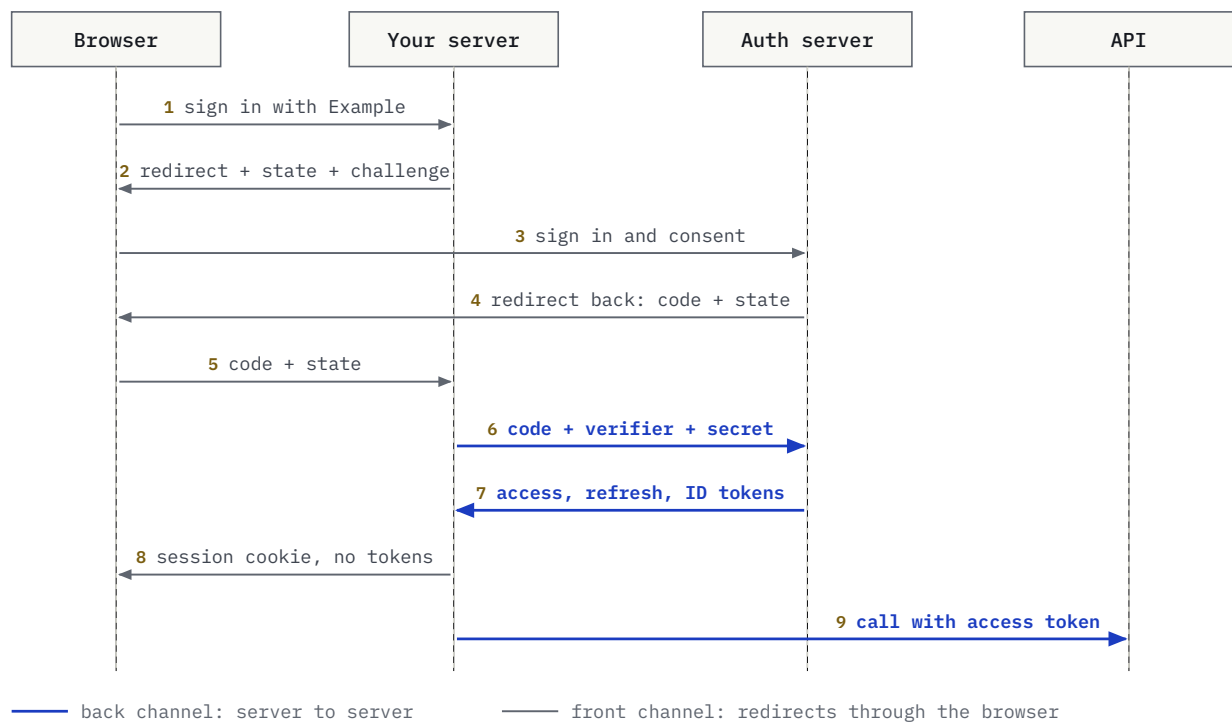
OAuth lets one application act on a user's behalf at another without ever seeing their password. OpenID Connect adds a standard way to learn who the user is. Use the authorization code flow with PKCE, keep tokens out of the browser, and validate everything you receive.

The problem OAuth solves

Before OAuth, an app that wanted your contacts asked for your email password, and then had full, permanent access to your account. OAuth replaces that with a limited, revocable grant: the user approves specific permissions, called **scopes**, at the provider, and your app receives an **access token** that can do only those things.

Four parties are involved. The **user** owns the data. The **client** is your app. The **authorization server** authenticates the user and issues tokens; think of Google's or Microsoft's sign-in. The **resource server** is the API holding the data. OAuth on its own is about delegated *access*, not about who the user is. **OpenID Connect (OIDC)** adds an **ID token**, a signed statement of who signed in, and that's what "Sign in with Google" actually uses.

The authorization code flow, with PKCE



The browser only ever carries a short-lived, single-use code. Tokens change hands server to server in steps 6 and 7, and never reach the browser at all.

- 01 **Start.** Your server generates a random `state` and a random PKCE `code_verifier`, keeps both in the user's pre-login session, and redirects the browser to the authorization server with the verifier's SHA-256 hash, the `code_challenge`.
- 02 **Sign in.** The user authenticates at the provider and approves the requested scopes. Your app never sees their credentials.
- 03 **Return.** The provider redirects to your registered `redirect_uri` with a one-time `code` and the same `state`.
- 04 **Check and exchange.** Your server confirms `state` matches, then sends the code, the original verifier and its own client credentials straight to the provider's token endpoint.
- 05 **Validate.** It receives the tokens and validates the ID token's signature, issuer, audience, expiry and nonce.
- 06 **Sign in locally.** It creates its own session (chapter 5) and gives the browser only a session cookie.

Each ingredient blocks a specific attack. `state` stops an attacker completing a login in your user's browser with the attacker's own code. PKCE stops a stolen code being redeemed by anyone who doesn't also hold the verifier. The `nonce` ties an ID token to one login attempt so it can't be replayed. Exact matching of redirect URIs, with no wildcards, stops codes being delivered to an attacker's page.

EXAMPLE oauth.py · starting the flow

```
import base64
import hashlib
import secrets
from urllib.parse import urlencode

def pkce_pair() -> tuple[str, str]:
    verifier = secrets.token_urlsafe(64) # 86 characters
    digest = hashlib.sha256(verifier.encode()).digest()
    challenge = base64.urlsafe_b64encode(digest).decode().rstrip("=")
    return verifier, challenge

def authorization_url(session) -> str:
    verifier, challenge = pkce_pair()
    session["oauth_state"] = state = secrets.token_urlsafe(32)
    session["oauth_nonce"] = nonce = secrets.token_urlsafe(32)
    session["pkce_verifier"] = verifier
    return AUTHORIZE_URL + "?" + urlencode({
        "response_type": "code",
        "client_id": CLIENT_ID,
        "redirect_uri": "https://app.example.com/auth/callback",
        "scope": "openid email profile",
        "state": state,
        "nonce": nonce,
        "code_challenge": challenge,
        "code_challenge_method": "S256",
    })
```

EXAMPLE oauth.py · the callback

```
def callback(session, params, http):
    expected_state = session.pop("oauth_state", "")
    if not expected_state or not secrets.compare_digest(params.get("state", ""),
        expected_state):
        raise Forbidden("state mismatch")

    tokens = http.post(TOKEN_URL, data={
        "grant_type": "authorization_code",
        "code": params["code"],
        "redirect_uri": "https://app.example.com/auth/callback",
        "code_verifier": session.pop("pkce_verifier"),
        "client_id": CLIENT_ID,
        "client_secret": CLIENT_SECRET,
    }, timeout=10).json()

    claims = verify_id_token(tokens["id_token"], issuer=ISSUER, audience=CLIENT_ID,
        nonce=session.pop("oauth_nonce"))
    user = find_or_link_user(issuer=claims["iss"], subject=claims["sub"],
        email=claims.get("email"),
        email_verified=claims.get("email_verified", False))
    return start_new_session(user)
```

Note how the user is identified: by the pair of issuer and subject, `iss` and `sub`, which the provider guarantees are stable and unique. **Never identify users by the email claim.** Some providers let users set an email address they haven't verified, and some allow it to change.

Apps that matched accounts by email have been taken over by attackers who simply registered a victim's address at a provider. Link a provider identity to an existing account by email only when the provider marks the email as verified, and preferably only after the user has also signed in to the existing account.

The tokens

TOKEN	WHAT IT IS	TYPICAL LIFETIME	WHO USES IT
Authorization code	A single-use voucher	Seconds to a minute	Your server, to exchange for tokens
Access token	Permission to call an API, within its scopes	Minutes to an hour	The resource server
Refresh token	Gets new access tokens without asking the user again	Days to months; rotated on each use	The authorization server
ID token	A signed statement of who signed in	Minutes	Your app, once, at sign-in

Never send an ID token to an API as if it were an access token, and never accept an access token issued for a different audience. With refresh-token rotation, each refresh returns a new refresh token and invalidates the old one; if an old one is ever presented again, someone has copied it, so revoke the whole family.

What's deprecated, and why

RFC 9700, the OAuth 2.0 Security Best Current Practice published in January 2025, retired two flows you'll still find in old tutorials. The **implicit grant** returned access tokens directly in the redirect URL, where browser history, logs and referrer headers could leak them. The **resource owner password grant** had the app collect the user's password, defeating the point of OAuth. It also requires PKCE for every authorization code flow. The in-progress OAuth 2.1 consolidates these rules into one document.

Tokens in browser apps

The IETF's guidance for browser-based applications, now published as RFC 10017, prefers a **backend-for-frontend**: your own server runs the flow, holds the tokens, and gives the browser nothing but an `HttpOnly` session cookie, exactly as in the diagram. If a single-page app must hold tokens itself, keep them in memory only, never in `localStorage` or `sessionStorage`, where every script on the page, including third-party ones, can read them.

When you're the provider

If your product offers an API to other companies, you become the authorization server. Use a proven implementation rather than writing one. Design scopes around what integrations actually need, such as `invoices:read` and `payments:write`, and show them plainly on the consent screen. Let customers see and revoke every connected app.

For server-to-server customers, API keys are often simpler than OAuth. Treat them like passwords: show a key once, store only its hash, allow several per account so they can be rotated without downtime, scope them, and give them a recognisable prefix such as `ik_live_`. That lets secret-scanning tools, including those run by code hosts, spot a key accidentally committed to a public repository.

COMMON TRAPS

OAuth as login without OIDC. An access token proves access, not identity. Validate an ID token.

Users matched by email. Key them by issuer and subject.

Skipping `state` or PKCE. Each blocks a real attack.

Loose redirect URI matching. Wildcards and prefix matching deliver codes to attackers.

Tokens in localStorage. Readable by any script on the page.

Long-lived access tokens with no rotation and no way to revoke them.

EXERCISES

EXERCISE 6.1 Name the defence

For each safeguard, name the attack it prevents.

- Checking that `state` in the callback matches the value stored before the redirect.
- Sending `code_verifier` when exchanging the code.
- Checking the ID token's `nonce`.
- Requiring an exact match on `redirect_uri`.
- Checking the ID token's `aud`.

SOLUTION

- a. **Login CSRF.** An attacker can't make your user's browser complete a sign-in with the attacker's own code, which would silently log your user into the attacker's account.
- b. **Code interception.** A code stolen from a redirect, a log or a malicious app is useless without the verifier, which never left your server.
- c. **ID token replay.** A token captured from one sign-in can't be used to complete another.
- d. **Code theft by redirect.** The provider won't send codes to a URL an attacker controls, such as an open redirect on a subdomain.
- e. **Token confusion.** A token issued to some other app, perhaps a malicious one the user also signed in to, can't be presented to yours.

EXERCISE 6.2 Find the takeover

This callback signs users in with any of several identity providers. Describe how an attacker takes over an existing account, and fix it.

```
def find_or_create_user(claims):
    user = db.find_user_by_email(claims["email"])
    if user:
        return user  # existing account: sign them in
    return db.create_user(email=claims["email"], name=claims.get("name"))
```

SOLUTION

The attacker signs up at any supported provider that doesn't verify email addresses, or lets them change one, and sets the victim's address. Signing in to your app with that identity returns the victim's account.

```
def find_or_link_user(claims):
    identity = db.find_identity(issuer=claims["iss"], subject=claims["sub"])
    if identity:
        return identity.user  # a known provider
                               account

    email, verified = claims.get("email"), claims.get("email_verified")
    existing = db.find_user_by_email(email) if email else None
    if existing:
        if not verified:
            raise LinkRequiresVerification()
        # Safer still: ask the user to sign in to the existing account first,
        # then link this provider from their settings page.
        raise AccountExistsSignInToLink(email)

    user = db.create_user(email=email if verified else None, name=claims.get("name"))
    db.create_identity(user_id=user.id, issuer=claims["iss"], subject=claims["sub"])
    return user
```

EXERCISE 6.3 Design the scopes

An accounting product wants to integrate with the invoicing app. It needs to read invoices and record payments made by bank transfer. Design the scopes it should request, and the consent screen's wording.

SOLUTION

- Request `invoices:read` and `payments:write`. Not `invoices:write`, since it never creates or edits invoices, and nothing touching team members or billing.
- Issue tokens per team, so granting access to one studio's books doesn't expose another's.
- Consent wording in plain language: "Acme Accounting wants to: see your invoices and their status; record payments against your invoices. It won't be able to create, edit or delete invoices." Then an Allow and a Cancel button.
- A settings page listing connected apps, their scopes, last use, and a revoke button.

CHECKLIST

- ☐ Authorization code flow with PKCE, `state`, `nonce` and exact redirect URIs?
- ☐ ID tokens validated for signature, issuer, audience, expiry and nonce?
- ☐ Users keyed by issuer and subject, not by email?
- ☐ Tokens held on the server, or in memory, and never in localStorage?
- ☐ Refresh tokens rotated, access tokens short-lived, and scopes minimal?

GO DEEPER

SPEC RFC 9700: OAuth 2.0 Security Best Current Practice

GUIDE OAuth 2.0 for Browser-Based Apps oauth.net's summary of RFC 10017

BOOK OAuth 2.0 Simplified Aaron Parecki

SPEC OpenID Connect Core 1.0

Authorisation and multi-tenancy

3 exercises

Authentication says who someone is; authorisation says what they may do, and to which object. Check it on every request, for every object, in one place, and make it impossible to query across customers by accident.

The most common serious bug in APIs

The OWASP API Security Top 10 has ranked the same flaw first since its first edition in 2019: **broken object level authorisation**, often called IDOR, for insecure direct object reference. The handler checks that someone is logged in, but not that the object they asked for is theirs. An attacker signs up, notices their invoice is `inv_1043`, and asks for `inv_1042`.

BEFORE any logged-in user can read any invoice

```
@app.get("/invoices/{invoice_id}")
def get_invoice(invoice_id: str, user=Depends(current_user)):
    return db.get_invoice(invoice_id)
```

AFTER scoped by tenant, then checked by policy

```
@app.get("/invoices/{invoice_id}")
def get_invoice(invoice_id: str, user=Depends(current_user)):
    invoice = db.get_invoice(invoice_id, team_id=user.team_id) # scoped query
    if invoice is None:
        raise NotFound() # the same answer for "doesn't exist" and "not yours"
    authorize(user, "invoice:read", invoice)
    return invoice
```

Two relatives appear in the same list. **Broken object property level authorisation** is a user sending a field they shouldn't control, such as `{"role": "owner"}` or `{"team_id": "..."}` , and the server saving it; this is also called mass assignment. **Broken function level authorisation** is an admin-only endpoint that a regular user can call because only the admin screen, not the endpoint, was protected.

Three models

MODEL	DECIDES USING	EXAMPLE RULE	FITS
Role-based (RBAC)	The user's role	"Admins can delete invoices."	Most products, most of the time

Attribute-based (ABAC)	Attributes of the user, the object and the context	"Members can edit invoices only while they're drafts."	Rules that depend on data
Relationship-based (ReBAC)	A graph of who relates to what	"You can view a document if you belong to a team that owns its folder."	Sharing, nested workspaces, documents

Most products need roles plus a few attribute rules. Relationship-based systems, modelled on Google's Zanzibar paper and available as open-source and hosted services, earn their complexity when users share individual objects with each other across a hierarchy.

One policy, in one place

Checks such as `if user.role == "admin"` scattered through handlers drift apart, and one endpoint always gets forgotten. Put every decision in a single policy module, deny by default, and test it exhaustively.

EXAMPLE `policy.py`

```
ROLE_PERMISSIONS = {
    "owner": {"invoice:read", "invoice:write", "invoice:delete", "team:manage",
             "billing:manage"},
    "admin": {"invoice:read", "invoice:write", "invoice:delete", "team:manage"},
    "member": {"invoice:read", "invoice:write"},
    "viewer": {"invoice:read"},
}

def can(user, action: str, invoice=None) -> bool:
    if invoice is not None and invoice.team_id != user.team_id:
        return False # tenancy first, always
    if action not in ROLE_PERMISSIONS.get(user.role, set()):
        return False # unknown roles get nothing
    if action == "invoice:write" and invoice is not None and invoice.status != "draft":
        return user.role in ("owner", "admin") # sent invoices: admins only
    return True

def authorize(user, action: str, invoice=None) -> None:
    if not can(user, action, invoice):
        raise Forbidden(action)
```

Your interface should hide the buttons a user can't use, because that's good design. But the interface is never the check. Every request is checked again on the server, because anyone can send any request with curl.

Multi-tenancy

In a business product, each customer organisation is a **tenant**: a team, a workspace, a studio. Every row belongs to one tenant, and one customer seeing another's data is the kind of incident that ends companies. There are four ways to isolate tenants.

ISOLATION	HOW	STRENGTHS	COSTS
Shared tables, tenant column	<code>WHERE team_id = ...</code> on every query	Cheapest and simplest to run	One missing clause leaks data
Shared tables with row-level security	The database enforces the tenant filter	A backstop when application code forgets	Setup care; must not connect as a privileged role
Schema per tenant	One set of tables per customer	Stronger isolation; per-tenant restores	Migrations run once per tenant
Database per tenant	A separate database per customer	Strongest isolation; data residency; no noisy neighbours	Most expensive to operate

A sensible default is shared tables with a tenant column, a data-access layer that adds the tenant filter automatically, and PostgreSQL row-level security underneath as a safety net.

EXAMPLE row-level security in PostgreSQL

```
ALTER TABLE invoices ENABLE ROW LEVEL SECURITY;
ALTER TABLE invoices FORCE ROW LEVEL SECURITY;      -- applies to the table owner too

CREATE POLICY tenant_isolation ON invoices
  USING      (team_id = current_setting('app.team_id', true)::uuid)
  WITH CHECK (team_id = current_setting('app.team_id', true)::uuid);

-- At the start of every request's transaction, from the authenticated session:
SET LOCAL app.team_id = '5b1d0e1e-6f3a-4c2e-9d61-0f1b2c3d4e5f';
```

If the setting has never been set on a connection, `current_setting(..., true)` returns null, the comparison fails, and the query sees no rows. On a pooled connection where an earlier request set it, the setting goes back to an empty string when that transaction ends, and casting it to `uuid` raises an error instead. Both are safe failures: no request sees another team's rows. `SET LOCAL` lasts only until the end of the transaction, which is what stops one request's team from carrying over to the next request on the same pooled connection. The application must connect as a role that doesn't own the tables and doesn't have the `BYPASSRLS` attribute. And background jobs, admin tools and reports need the tenant context too; they're where scoping is most often forgotten.

Mass assignment

Never copy a request body straight onto a model. Define an explicit input schema for each action, listing the only fields a caller may set, and reject anything else.

BEFORE

```
invoice.update(**request_json)    # a client can set team_id, status, paid_at...
```

AFTER with Pydantic

```
from datetime import date
from pydantic import BaseModel, ConfigDict, Field

class InvoiceUpdate(BaseModel):
    model_config = ConfigDict(extra="forbid")    # unknown fields are an error
    due_date: date | None = None
    notes: str | None = Field(default=None, max_length=2000)

changes = InvoiceUpdate.model_validate(request_json)
invoice.update(**changes.model_dump(exclude_unset=True))
```

Audit logs and impersonation

In any product used by teams, record who did what, to which object, and when: actor, tenant, action, object type and id, timestamp, and a little context. Keep the log append-only. Customers will ask for it, support needs it to answer "who deleted this?", and security needs it after an incident.

Support staff will eventually need to see what a customer sees. Build impersonation deliberately: explicit, time-limited, read-only by default, recorded in the audit log, and ideally visible to the customer. A hidden "log in as anyone" switch is a breach waiting for an insider.

COMMON TRAPS

Logged in, therefore allowed. Authentication without object-level checks.

Checks only in the interface. The API is the real front door.

403 that confirms existence. For private objects, "not yours" should look like "not found".

Request bodies bound to models. Mass assignment of roles, owners and prices.

Jobs and reports without tenant scope. The code path nobody reviews for authorisation.

Unaudited god mode. Admin powers with no record of their use.

EXERCISES

EXERCISE 7.1 Find the authorisation bugs

Each endpoint has one serious authorisation flaw. Name it and fix it.

```
@app.get("/invoices")
def list_invoices(team_id: str, user=Depends(current_user)):      # A
    return db.list_invoices(team_id=team_id)

@app.delete("/invoices/{invoice_id}")
def delete_invoice(invoice_id: str, user=Depends(current_user)):  # B
    if user.role not in ("owner", "admin"):
        raise Forbidden()
    db.delete_invoice(invoice_id)

@app.patch("/me")
def update_profile(body: dict, user=Depends(current_user)):      # C
    db.update_user(user.id, **body)
```

SOLUTION

- **A:** the tenant comes from the caller. Anyone can pass another team's id. Take it from the session:
`db.list_invoices(team_id=user.team_id)` .
- **B:** the role is checked, but not the invoice's team. An admin of one team can delete another team's invoices. Load the invoice scoped to `user.team_id` and call `authorize(user, "invoice:delete", invoice)` .
- **C:** mass assignment. A user can send `{"role": "owner"}` or `{"team_id": "..."} . Parse the body into a schema allowing only, say, name and timezone , with unknown fields forbidden.`

EXERCISE 7.2 Test the policy

Write a table-driven test for `can` above that covers cross-team access, each role, invoice status, and an unknown role.

SOLUTION

```
@pytest.mark.parametrize("role, action, invoice, allowed", [
    ("admin", "invoice:read", Invoice(team_id="other"), False), # never cross
    teams
    ("viewer", "invoice:read", Invoice(), True),
```

```

    ("viewer", "invoice:write", Invoice(), False),
    ("member", "invoice:write", Invoice(status="draft"), True),
    ("member", "invoice:write", Invoice(status="sent"), False),
    ("admin", "invoice:write", Invoice(status="sent"), True),
    ("member", "invoice:delete", Invoice(), False),
    ("owner", "billing:manage", None, True),
    ("intern", "invoice:read", Invoice(), False), # unknown
    role: nothing
])
def test_policy(role, action, invoice, allowed):
    user = User(role=role, team_id="team_1")
    assert can(user, action, invoice) is allowed

```

Here `Invoice()` defaults to `team_id="team_1"` and `status="draft"`. A table like this doubles as documentation: a new engineer can read the whole permission model in one screen.

EXERCISE 7.3 Choose the isolation

Choose a tenancy model for each customer, and justify it.

- Thousands of small studios that sign up by themselves.
- A hospital group whose contract requires its data to stay in its own region, with separate backups.
- An agency with five million invoices whose reports slow everyone else down.

SOLUTION

- Shared tables with row-level security.** Cheap per tenant, one set of migrations, and a database backstop.
- A dedicated database, or a whole dedicated deployment, in the required region.** Residency and separate backups are contractual requirements that shared tables can't satisfy.
- Shared tables, with this tenant moved to its own database or replica** once it's measurably a noisy neighbour. Make your data layer tenant-aware early, so moving one tenant is configuration rather than a rewrite. Heavy reports can also run against a read replica.

CHECKLIST

- ☐ Does every endpoint check access to the specific object, not just login?
- ☐ Is the policy in one module, denying by default, with table-driven tests?

- ☐ Is every query tenant-scoped, with a database-level backstop such as row-level security?
- ☐ Are request bodies parsed into explicit schemas for each action?
- ☐ Are sensitive actions and impersonation recorded in an audit log?

GO DEEPER

GUIDE OWASP API Security Top 10, 2023

GUIDE Authorization Cheat Sheet OWASP

DOCS Row Security Policies PostgreSQL

PAPER Zanzibar: Google's Consistent, Global Authorization System

PART III

The frontend

As far as users are concerned, the interface is the product. Four chapters: the platform underneath every framework, accessibility as a requirement rather than a feature, components and state, and making it all fast on a real phone.

The web platform

3 exercises

Frameworks come and go; HTML, CSS, the DOM and the event loop are what all of them compile down to. Learn the platform first, and every framework becomes easier to use well.

Semantic HTML does work for you

Use each element for its meaning: `<button>` for actions, `<a href>` for navigation, `<form>` and `<label>` for input, `<nav>` and `<main>` for landmarks, headings in order, `<table>` for tabular data. The browser attaches behaviour to meaning. A native button is focusable, activates with Enter and Space, announces itself to screen readers, supports a disabled state, and submits its form, all for free. A `div` with a click handler does none of it.

BEFORE

```
<div class="btn" onclick="sendInvoice()">Send invoice</div>
```

AFTER

```
<button type="button" onclick="sendInvoice()">Send invoice</button>
```

The rule of thumb: if it changes something, it's a button; if it goes somewhere, it's a link.

Forms: the most important interface you'll build

Sign-up, sign-in, checkout, settings: the moments that matter most in a product are forms. The platform gives you a lot if you let it.

- A visible `<label>` for every input, tied to it with `for` and `id`. A placeholder is not a label; it disappears as soon as someone types.
- The right `type`: `email`, `tel`, `url`, `date`. They bring the right mobile keyboard and built-in checks.
- An `autocomplete` token such as `email`, `username`, `current-password`, `new-password`, `one-time-code` or `postal-code`. Password managers and autofill depend on them, and they save your users more time than any feature you'll build this month.
- Native constraints such as `required` and `minlength` as a first line of feedback. The server's validation is the one that counts.

- **Errors next to the field**, in words, connected with `aria-describedby`, and never by clearing what the user typed.

EXAMPLE a sign-in form that browsers and password managers understand

```
<form method="post" action="/login">
  <label for="email">Email</label>
  <input id="email" name="email" type="email" autocomplete="username" required>

  <label for="password">Password</label>
  <input id="password" name="password" type="password"
    autocomplete="current-password" required
    aria-describedby="login-error">

  <p id="login-error" class="field-error" hidden>Email or password is incorrect.</p>

  <button type="submit">Sign in</button>
</form>
```

CSS layout: flexbox and grid

Set `box-sizing: border-box` everywhere, so padding and borders sit inside an element's width instead of adding to it. Then use two layout tools. **Flexbox** lays items out along one axis: a toolbar, a row of buttons, a label beside its value. **Grid** lays items out in two dimensions: page structure, card grids, forms with aligned columns. Positioning things with absolute offsets and magic numbers is almost always a sign one of these was the right tool.

Design for the smallest screen first, and let layouts grow. **Container queries** let a component respond to the space it's actually given rather than to the whole window, so the same invoice card works in a narrow sidebar and a wide list.

EXAMPLE cards that fill the space, and adapt to their own width

```
.invoice-grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(min(18rem, 100%), 1fr));
  gap: 1rem;
}

.invoice-card {
  container-type: inline-size; /* children can query this card's width */
}

@container (min-width: 26rem) {
  .invoice-card .meta {
    display: flex;
    justify-content: space-between; /* side by side only when there's room */
  }
}
```

The DOM and events

The DOM is the live tree of objects the browser builds from your HTML, and every framework ultimately reads and writes it. Two event concepts are worth knowing cold. Events **bubble** from the element where they happened up through its ancestors, so one listener on a list can handle clicks on every row, including rows added later. That's **event delegation**. And `event.target` is where the event started, while `event.currentTarget` is the element whose listener is running.

EXAMPLE one listener for a whole list

```
document.querySelector("#invoice-list").addEventListener("click", (event) => {
  const button = event.target.closest("button[data-action='remind']");
  if (!button) return; // a click somewhere else in the list
  sendReminder(button.dataset.invoiceId);
});
```

The event loop

Each page runs its JavaScript on one main thread, which is also the thread that responds to input and paints the screen. Work arrives as **tasks**: a click handler, a timer, a network callback. After each task, the browser runs every queued **microtask**, which is where promise callbacks go, and then it may render. So a task that runs for 300 milliseconds freezes the page for 300 milliseconds: clicks wait, typing lags, nothing repaints. The browser calls anything over 50 milliseconds a long task, and long tasks are the main cause of poor responsiveness scores (chapter 11).

EXAMPLE run it and check the order

```
console.log("1: script");
setTimeout(() => console.log("4: timer task"), 0);
Promise.resolve().then(() => console.log("3: microtask"));
console.log("2: script end");
// Prints 1, 2, 3, 4. Microtasks run before the next task, even a 0 ms timer.
```

To keep the page responsive, break long work into chunks and yield to the browser between them, move heavy computation to a Web Worker, or do it on the server. Where it's supported, `await scheduler.yield()` is the cleanest way to give the browser a chance to handle input mid-task.

Progressive enhancement

Start with HTML that works on its own: a form that posts to the server and gets a page back. Then enhance it with JavaScript for a smoother experience. JavaScript fails more often than developers think: flaky networks, browser extensions, a script error on an old device, a corporate proxy. When it fails, an enhanced form still works, and a JavaScript-only form is a blank rectangle. Server-rendered forms with a layer of enhancement are also, very often, the fastest thing you can ship.

COMMON TRAPS

Div soup. Clickable `div` s instead of buttons and links.

Placeholders as labels. Gone the moment someone types, and often too faint to read.

Blocking paste in password fields. It breaks password managers and pushes people towards weak passwords.

`type="number"` for things that aren't quantities. Card numbers and postcodes lose leading zeros and gain spinner arrows.

Layout by magic numbers. Absolute positioning that works on one screen size.

Long tasks on the main thread. The page looks loaded and ignores the user.

EXERCISES

EXERCISE 8.1 Fix the billing form

Rewrite this billing-details form so browsers, password managers and assistive technology understand it.

```
<div class="form">
  <input placeholder="Full name">
  <input placeholder="Email">
  <input type="number" placeholder="Postcode">
  <input type="number" placeholder="Phone">
  <div class="btn" onclick="save()">Save</div>
</div>
```

SOLUTION

```
<form method="post" action="/settings/billing">
  <label for="name">Full name</label>
  <input id="name" name="name" autocomplete="name" required>

  <label for="email">Billing email</label>
  <input id="email" name="email" type="email" autocomplete="email" required>

  <label for="postcode">Postcode</label>
  <input id="postcode" name="postcode" autocomplete="postal-code">

  <label for="phone">Phone</label>
  <input id="phone" name="phone" type="tel" autocomplete="tel">

  <button type="submit">Save billing details</button>
</form>
```

Postcodes and phone numbers are text, not numbers: some start with zero, some contain letters, spaces or plus signs, and none should have spinner arrows. The button now submits a real form, so it works before JavaScript loads, and pressing Enter submits it too.

EXERCISE 8.2 Predict the order

In what order are the letters printed? Work it out before running it.

```
console.log("A");
setTimeout(() => console.log("B"), 0);
queueMicrotask(() => console.log("C"));
(async () => {
  console.log("D");
  await null;
  console.log("E");
})();
Promise.resolve().then(() => console.log("F"));
console.log("G");
```

SOLUTION

A D G C E F B. The synchronous code runs first: A, then D, because an async function runs synchronously until its first `await`, then G. The microtasks then run in the order they were queued: C, then the rest of the async function, E, then F. Only after the microtask queue is empty does the timer task run: B.

EXERCISE 8.3 Lay out the page

Write CSS for an app layout with a header, a 16rem sidebar and a main area. On screens narrower than 48rem, the sidebar should move below the header and span the full width.

SOLUTION

```
.app {
  display: grid;
  grid-template-areas:
    "header"
    "sidebar"
    "main";
  min-height: 100dvh;
}

@media (min-width: 48rem) {
  .app {
    grid-template-columns: 16rem minmax(0, 1fr);
    grid-template-areas:
      "header header"
```

```
    "sidebar main";
  }
}

.app > header { grid-area: header; }
.app > nav    { grid-area: sidebar; }
.app > main   { grid-area: main; }
```

Mobile first: the base rule is the narrow layout, and the media query adds columns when there's room. `minmax(0, 1fr)` stops a wide table or code block inside `main` from forcing the column wider than the screen.

CHECKLIST

- ☐ Does every interactive element use the native element for its job?
- ☐ Does every input have a visible label, the right type and an autocomplete token?
- ☐ Is layout done with flexbox and grid, and does it work from 320 pixels wide?
- ☐ Is any main-thread task long enough to block input?
- ☐ Does the core flow work before JavaScript loads?

GO DEEPER

COURSE [Learn web development MDN](#)

COURSE [Learn Forms web.dev](#)

COURSE [Learn CSS web.dev](#)

ESSAY [Tasks, microtasks, queues and schedules Jake Archibald](#)

Accessibility

3 exercises

Accessibility means people with disabilities can use what you build, whether by keyboard, screen reader, magnification or voice. Build it in from the first component, because retrofitting is expensive, and in more and more markets it's the law.

Why it matters

The World Health Organization estimates that about 1.3 billion people, one in six, live with a significant disability. Add temporary and situational limits, such as a broken wrist, a toddler on one arm, or sunlight on a screen, and accessibility stops being about a minority of users.

The web is still doing badly. WebAIM tests the home pages of the top million websites every year. In its 2026 report, 95.9% had detectable failures of the Web Content Accessibility Guidelines, an average of 56 errors per page and worse than the year before. The encouraging part is that six kinds of error account for 96% of everything found, and all six are straightforward to fix.

FAILURE	HOME PAGES AFFECTED	THE FIX
Low-contrast text	83.9%	At least 4.5:1 for normal text, 3:1 for large text and interface parts
Images without alt text	53.1%	Describe what the image means; <code>alt=""</code> if it's purely decorative
Form inputs without labels	51.0%	A <code><label></code> tied to every input
Empty links	46.3%	Give icon-only links an accessible name
Empty buttons	30.6%	Give icon-only buttons an accessible name
No document language	13.5%	<code><html lang="en"></code>

The law is catching up. The European Accessibility Act has applied since 28 June 2025 to e-commerce and many other consumer services sold in the EU, including by companies based elsewhere, and it measures conformance against a standard built on WCAG. In the

United States, lawsuits over inaccessible websites run into the thousands every year. The usual target is WCAG 2.2, level AA.

The four principles

WCAG organises everything under four principles, known by the acronym POUR. Content must be **perceivable**: available to at least one sense a user has, so images have text alternatives and videos have captions. The interface must be **operable**: usable with a keyboard, with enough time, without triggering seizures. It must be **understandable**: predictable, readable, and helpful when mistakes happen. And it must be **robust**: built with standard markup that assistive technologies can interpret.

Accessible names

Every interactive element needs a name that a screen reader can announce. Text inside a button is its name. An icon has no name unless you give it one.

BEFORE announced as "button"

```
<button class="icon-btn">
  <svg viewBox="0 0 24 24"><path d="M6 7h12M9 7V4h6v3m-8 0 1 13h8l1-13"/></svg>
</button>
```

AFTER announced as "Delete invoice INV-0042, button"

```
<button class="icon-btn" aria-label="Delete invoice INV-0042">
  <svg viewBox="0 0 24 24" aria-hidden="true" focusable="false">
    <path d="M6 7h12M9 7V4h6v3m-8 0 1 13h8l1-13"/>
  </svg>
</button>
```

Note the specific name. A table with fifty rows, each with a button called "Delete", is fifty identical announcements. Including the invoice number tells a screen reader user which invoice they're about to delete.

Keyboard and focus

Everything must work with a keyboard alone: Tab to move, Enter and Space to activate, Escape to dismiss. The tab order follows the order of elements in the HTML, so keep the source order logical rather than rearranging things visually with CSS. Every focused element needs a clearly visible indicator. Never remove the outline without replacing it; `:focus-visible` lets you show it for keyboard users without showing it on every mouse click.

Single-page apps must manage focus themselves, because they change the page without the browser's usual navigation. After a route change, move focus to the new page's main heading. When a dialog opens, focus moves into it and stays there until it closes, and then returns to whatever opened it. The native `<dialog>` element, opened with `showModal()`, does most of this for you: it traps focus, closes on Escape, and makes the rest of the page inert.

EXAMPLE a confirmation dialog with correct focus behaviour, for free

```
<button id="open-send" type="button">Send invoice</button>

<dialog id="send-dialog" aria-labelledby="send-title">
  <h2 id="send-title">Send INV-0042 to Studio North?</h2>
  <form method="dialog">
    <button value="cancel">Cancel</button>
    <button value="send">Send invoice</button>
  </form>
</dialog>

<script>
  const dialog = document.getElementById("send-dialog");
  document.getElementById("open-send").addEventListener("click", () => dialog.showModal());
  dialog.addEventListener("close", () => {
    if (dialog.returnValue === "send") sendInvoice("INV-0042");
  });
</script>
```

ARIA: the first rule

ARIA attributes add roles, states and names that HTML can't express on its own. The W3C's first rule of ARIA is not to use it when a native element already does the job. WebAIM found that home pages using ARIA had noticeably more detected errors on average than those without it. That's a correlation, not proof of cause, but it matches experience: misapplied ARIA actively makes pages worse, because it overrides what the browser would have announced correctly. The ARIA Authoring Practices Guide puts it bluntly: no ARIA is better than bad ARIA.

Where ARIA earns its place: announcing changes that happen without a page load, such as "Invoice sent", through a live region; the state of disclosure buttons with `aria-expanded`; the current page in navigation with `aria-current="page"`; and linking inputs to their error messages with `aria-describedby` and `aria-invalid`.

EXAMPLE announcing an update without moving focus

```
<div id="status" role="status"></div>

<script>
  function announce(message) {
    document.getElementById("status").textContent = message;
  }
  announce("Invoice INV-0042 sent to Studio North.");
</script>
```

What WCAG 2.2 added

The 2023 revision added criteria that catch common modern mistakes. Focused elements must not be completely hidden behind sticky headers or cookie banners. Click targets must be at least 24 by 24 CSS pixels, or spaced out enough to avoid mis-taps. Anything done by dragging needs a single-pointer alternative. Help must appear in a consistent place. Users shouldn't have to re-enter information they've already given. And sign-in must not depend on a cognitive test, such as retyping characters or solving a puzzle, without an alternative; allowing paste and password managers, as chapter 5 recommends, helps satisfy it.

Testing

Automated checkers such as axe-core and Lighthouse find the mechanical problems quickly, and they belong in your CI pipeline. They can't tell you whether alt text is meaningful, whether the focus order makes sense, or whether a flow is usable at all, so pair them with manual checks.

EXAMPLE `invoice.a11y.spec.ts`, with Playwright and axe

```
import { test, expect } from "@playwright/test";
import AxeBuilder from "@axe-core/playwright";

test("invoice page has no detectable accessibility violations", async ({ page }) => {
  await page.goto("/invoices/INV-0042");
  const results = await new AxeBuilder({ page })
    .withTags(["wcag2a", "wcag2aa", "wcag22aa"])
    .analyze();
  expect(results.violations).toEqual([]);
});
```

- **Put the mouse away** and complete every key flow with the keyboard: sign up, create an invoice, send it, pay it.
- **Use a screen reader** for a few minutes on each new component: VoiceOver on macOS and iOS, NVDA on Windows, TalkBack on Android.
- **Zoom to 200% and 400%**, and check that nothing overlaps or disappears and that you rarely need to scroll sideways.
- **Turn on reduced motion and high contrast** in the operating system, and check that your interface respects them.

COMMON TRAPS

Accessibility as a final audit. By then, the inaccessible patterns are in every component.

Overlay widgets. Scripts that promise automatic compliance don't deliver it. In 2025 the US

Federal Trade Commission fined one such vendor over claims that its tool made sites compliant.

Removing focus outlines without providing a replacement.

Colour as the only signal. A red dot for "overdue" says nothing to someone who can't distinguish red.

Custom widgets built from `div` s with no keyboard support.

ARIA sprinkled on to silence a checker rather than to describe real behaviour.

EXERCISES

EXERCISE 9.1 Fix the invoice row

This row in the invoice list has four accessibility problems. Find them and rewrite it.

```
<div class="row" onclick="openInvoice('INV-0042')">
  <span class="dot red"></span>
  <span style="color:#b5b5b5">INV-0042 · Studio North · €1,250.00</span>
  <div class="menu-btn" onclick="openMenu(event)"><span></div>
</div>
```

SOLUTION

1. The whole row is a clickable `div` : no keyboard access and no role.
2. The status is conveyed only by a red dot.
3. The grey text is far below 4.5:1 contrast on white.
4. The menu "button" is a `div` whose only content is an ellipsis character.

```
<tr>
  <th scope="row"><a href="/invoices/INV-0042">INV-0042</a></th>
  <td>Studio North</td>
  <td class="amount">€1,250.00</td>
  <td><span class="badge badge-overdue">Overdue</span></td>
  <td>
    <button type="button" aria-haspopup="menu" aria-expanded="false"
      aria-label="More actions for INV-0042"><span></button>
  </td>
</tr>
```

The invoice number becomes a real link, the status is written in words, the text uses a colour with enough contrast, and the menu is a real button with a specific name that reports whether it's open. It's also now a table row, because this is tabular data.

EXERCISE 9.2 Walk it with a keyboard

Describe, step by step, what should happen when a keyboard user sends an invoice from its detail page, through a confirmation dialog, and back.

SOLUTION

1. Tab reaches "Send invoice" in a logical position, with a visible focus ring.
2. Enter opens the dialog. Focus moves into it, onto the first control or the heading, and the screen reader announces the dialog's title.
3. Tab cycles only among the dialog's controls. The page behind can't be reached.
4. Escape closes the dialog without sending, and focus returns to "Send invoice".
5. Choosing "Send invoice" closes the dialog, focus returns to the trigger, and a live region announces "Invoice INV-0042 sent to Studio North".
6. The status badge on the page updates to "Sent", in words.

EXERCISE 9.3 Name the icons

Write the accessible name, or the alt text, for each of these on the invoice detail page for INV-0042.

- a. A pencil icon button that opens the editor.
- b. A paper-plane icon button that sends the invoice.
- c. A bell icon button that configures payment reminders.
- d. The studio's uploaded logo, shown on the invoice preview.
- e. A decorative wave illustration in the page header.

SOLUTION

- a. `aria-label="Edit invoice INV-0042"`
- b. `aria-label="Send invoice INV-0042"`

- c. `aria-label="Payment reminders for INV-0042"`
- d. `alt="Brightline Studio logo"` , naming the freelancer's own studio, because on an invoice the logo identifies who sent it.
- e. `alt=""` . It carries no information, so screen readers should skip it entirely. Leaving `alt` out altogether is different, and worse: some screen readers then read the file name.

CHECKLIST

- ☐ Do text and controls meet the contrast minimums?
- ☐ Does every image, input, link and button have an accessible name?
- ☐ Can every flow be completed by keyboard, with visible focus?
- ☐ Is focus managed on route changes and in dialogs?
- ☐ Is ARIA used only where native HTML can't do the job?
- ☐ Are automated checks in CI, backed by a keyboard and screen-reader pass before release?

GO DEEPER

SPEC [How to Meet WCAG 2.2 the W3C's filterable quick reference](#)

REPORT [The WebAIM Million 2026 edition](#)

GUIDE [ARIA Authoring Practices Guide patterns for widgets HTML doesn't provide](#)

BOOK [Inclusive Components Heydon Pickering](#)

Components, state and data fetching

3 exercises

Most frontend complexity comes from state kept in the wrong place. Treat server data as a cache of something you don't own, keep interface state as local as possible, put shareable state in the URL, and derive everything else.

Components

A component is a function from its inputs, called props, and its own state, to a piece of interface. The deep-module thinking of *The Engineer's Craft* applies directly: a good component does one job, exposes a small, clear set of props, hides its internals, and has no hidden dependencies on global variables. Compose small components through their children rather than building one component with thirty configuration flags.

EXAMPLE StatusBadge.tsx

```
type InvoiceStatus = "draft" | "sent" | "paid" | "overdue" | "void";

const LABELS: Record<InvoiceStatus, string> = {
  draft: "Draft",
  sent: "Sent",
  paid: "Paid",
  overdue: "Overdue",
  void: "Void",
};

export function StatusBadge({ status }: { status: InvoiceStatus }) {
  // The status is always written in words; colour only reinforces it.
  return <span className={`badge badge-${status}`}>{LABELS[status]}</span>;
}
```

TypeScript's `Record<InvoiceStatus, string>` is doing quiet work here. Add a new status to the union and this file stops compiling until it has a label, which is exactly the reminder you want.

Four kinds of state

KIND	EXAMPLES IN THE INVOICING APP	WHERE IT BELONGS
Server state	Invoices, clients, the signed-in user	A query cache such as TanStack Query, or your framework's data loaders
URL state	Filters, sort order, page, selected tab, the open invoice	The path and query string

Local interface state	Is this menu open? What has been typed so far?	The component itself, with <code>useState</code>
Global client state	Theme; a draft spanning several screens of a wizard	A small store or context, and rarely needed

The React community's biggest shift of the last few years was recognising that server state and client state are different problems. Server data is a borrowed copy of something that lives elsewhere. It goes stale, other people change it, it must be refetched, deduplicated and invalidated. Putting it into a general-purpose global store means rebuilding a cache badly, by hand.

Derive, don't duplicate

If a value can be computed from other state, compute it. Every extra piece of state is something that can fall out of sync.

BEFORE three pieces of state kept in step by hand

```
const [invoices, setInvoices] = useState<Invoice[]>([]);
const [overdue, setOverdue] = useState<Invoice[]>([]);
const [overdueTotal, setOverdueTotal] = useState(0);
// ...and every update must remember to change all three
```

AFTER one source of truth, the rest derived on render

```
const { data: invoices = [] } = useInvoices();
const overdue = invoices.filter((invoice) => invoice.status === "overdue");
const overdueTotal = overdue.reduce((sum, invoice) => sum + invoice.amountDueMinor, 0);
```

Server state in a query cache

EXAMPLE invoices.queries.ts, with TanStack Query

```
import { useMutation, useQuery, useQueryClient } from "@tanstack/react-query";

export function useInvoices(status?: InvoiceStatus) {
  return useQuery({
    queryKey: ["invoices", { status }],
    queryFn: ({ signal }) => api.get<Invoice[]>("/invoices", { params: { status }, signal }),
    staleTime: 30_000, // reuse cached data for 30 seconds before refetching
  });
}

export function useSendInvoice() {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(api.post<Invoice>("/invoices", { params: { status }, signal }));
}
```

```

return useMutation({
  mutationFn: (id: string) => api.post(`/invoices/${id}/send`),
  onSuccess: () => queryClient.invalidateQueries({ queryKey: ["invoices"] }),
});
}

```

Two components asking for the same invoices share one request. Data refreshes in the background when the window regains focus. Failed requests retry. After sending an invoice, every cached invoice list is marked stale and refetched. None of that is code you write.

Every view that shows data has four states, not one: **loading**, **error**, **empty** and **data**. Design all four. The empty state of a brand-new account is the first thing a new user sees, and a blank table with no guidance is a common reason people never come back.

URL state

Filters, sort order and the selected tab belong in the URL. Then a view can be bookmarked, shared with a colleague, survives a refresh, and the back button does what people expect.

EXAMPLE a status filter in the query string, with React Router

```

function StatusFilter() {
  const [params, setParams] = useSearchParams();
  const status = params.get("status") ?? "all";

  return (
    <label>
      Status
      <select value={status}
        onChange={(e) => setParams((p) => { p.set("status", e.target.value); return p;
        }}>
        <option value="all">All</option>
        <option value="overdue">Overdue</option>
        <option value="paid">Paid</option>
      </select>
    </label>
  );
}

```

Forms

Keep form fields local until submission, validate on the client for fast feedback, and treat the server's answer as final. Where you can, share one validation schema between client and server so the rules can't drift apart. Show server errors next to the right fields, using the field paths from the problem details of chapter 1. React 19 added Actions for form submissions, which track pending and error state for you and cooperate with server-side rendering.

Optimistic updates

For quick, low-risk, almost-always-successful actions, such as archiving an invoice or renaming a client, update the interface immediately and roll back if the server refuses. It makes a product feel instant. Never do it for anything irreversible or financial: a payment must show "processing" until the server confirms it.

EXAMPLE archive now, undo on failure

```
const archive = useMutation({
  mutationFn: (id: string) => api.post(`/invoices/${id}/archive`),
  onMutate: async (id) => {
    await queryClient.cancelQueries({ queryKey: ["invoices"] });
    const previous = queryClient.getQueryData<Invoice[]>(["invoices", {}]);
    queryClient.setQueryData<Invoice[]>(["invoices", {}], (old = []) =>
      old.filter((invoice) => invoice.id !== id));
    return { previous };
  },
  onError: (_error, _id, context) => {
    queryClient.setQueryData(["invoices", {}], context?.previous);
    toast("Couldn't archive the invoice. It's back in the list.");
  },
  onSettled: () => queryClient.invalidateQueries({ queryKey: ["invoices"] }),
});
```

Where rendering happens

React Server Components, now the default in frameworks such as Next.js, let some components run only on the server: they fetch data directly and send rendered output, and none of their code ships to the browser. Components that need interactivity are marked as client components. A common pattern is server components for the initial page, a query cache for data that changes while the page is open, and local state for the rest. Chapter 11 looks at what this does for performance.

COMMON TRAPS

A global store used as a cache. Copying server data into it means hand-writing staleness, refetching and invalidation.

Derived values in state. They drift out of sync with their source.

Filters that live only in memory. Refresh the page and they're gone; share the link and they're gone.

Only designing the happy path. No loading, error or empty states.

Fetching in `useEffect` with no cache. Duplicate requests, and responses that arrive out of order.

Optimism about money Optimistic updates on payments or sends that can't be undone

EXERCISES

EXERCISE 10.1 Classify the state

For each piece of state, say which kind it is and where it should live.

- The signed-in user's name and role.
- The list of invoices on the dashboard.
- The text in the client search box, before the debounce fires.
- The "overdue only" filter on the invoice list.
- Whether the "new invoice" side panel is open.
- An invoice being built across a four-step wizard.
- The user's choice of dark mode.
- The total of all overdue invoices.

SOLUTION

- Server state, in the query cache, fetched once and shared.
- Server state, in the query cache.
- Local state in the search component.
- URL state: `?status=overdue`.
- Local state, or URL state if people should be able to link straight to the open panel.
- Global client state for the wizard's lifetime, or better, a draft saved to the server after each step so it survives a closed tab.
- A user preference: saved on the server if it should follow them between devices, otherwise in local storage.
- Not state at all. Derive it from the invoice list on render.

EXERCISE 10.2 Fix the race

Typing "stu" into the client search sometimes shows the results for "st". Find the two bugs, and fix them.

```
useEffect(() => {  
  fetch(`/api/clients?q=${query}`)  
    .then((response) => response.json())  
    .then(setResults);  
})
```

```
}, [query]);
```

SOLUTION

A race: the requests for "s", "st" and "stu" are all in flight at once, and if "st" answers last, its results overwrite the right ones. **An encoding bug:** a query containing `&` or `#` breaks the URL.

```
useEffect(() => {
  const controller = new AbortController();
  fetch(`/api/clients?q=${encodeURIComponent(query)}`, { signal: controller.signal })
    .then((response) => response.json())
    .then(setResults)
    .catch((error) => {
      if (error.name !== "AbortError") setError(error);
    });
  return () => controller.abort(); // cancel the previous request when query
    changes
}, [query]);
```

Or let the query cache do it: `useQuery({ queryKey: ["clients", query], queryFn: ({ signal }) => searchClients(query, signal) })` keys results by query, so a late answer for "st" can never be shown for "stu".

EXERCISE 10.3 Design the four states

Describe the loading, error, empty and data states for the invoice list. Treat "a brand-new account" and "a filter that matches nothing" as different empty states.

SOLUTION

- **Loading:** a skeleton with the same row layout, so nothing jumps when data arrives. Show it only after about 200 milliseconds, so fast loads don't flash.
- **Error:** "We couldn't load your invoices" with a Retry button. If older data is cached, keep showing it with a quiet notice instead of wiping the screen.
- **Empty, new account:** a short explanation and one clear action: "Create your first invoice". This is onboarding, not an error.
- **Empty, filtered:** "No overdue invoices", perhaps with good news in it, and a "Clear filter" button.
- **Data:** the list, with the filter reflected in the URL and the count visible.

CHECKLIST

- ☐ Is server data held in a query cache rather than copied into a global store?
- ☐ Are filters, sorting and selection in the URL?
- ☐ Is anything stored in state that could be derived?
- ☐ Does every data view handle loading, error, empty and data?
- ☐ Are optimistic updates limited to reversible, low-risk actions?

GO DEEPER

DOCS Thinking in React [react.dev](#)

DOCS You Might Not Need an Effect [react.dev](#)

DOCS TanStack Query

ESSAY Application State Management with React [Kent C. Dodds](#)

Rendering strategies and web performance

3 exercises

Where the HTML is produced, and how much JavaScript the user must download, decide how fast a product feels. Choose rendering page by page, measure real users with Core Web Vitals, and treat bytes as a budget.

Where rendering happens

STRATEGY	HTML IS PRODUCED	GOOD FOR	COST
Client-side rendering	In the browser, after JavaScript loads	App screens behind a login	Blank or skeleton until the bundle runs; large downloads
Server-side rendering	On the server, for each request	Personalised pages that must appear fast	Server work per request; hydration afterwards
Static generation	Once, at build time	Marketing pages, documentation, blogs	A rebuild to change anything
Incremental regeneration	Statically, refreshed on a schedule or on demand	Large catalogues	Content can be briefly stale
Streaming	On the server, sent in pieces as data arrives	Pages with one slow section	Needs framework support
Islands	Static HTML with small interactive regions	Content sites with a little interactivity	Less suited to highly interactive apps

Hydration is the hidden cost of server rendering. The HTML arrives quickly and looks ready, but the JavaScript for every component on the page must still download, run, and reattach event handlers before anything responds. Until then, a click does nothing. Server Components and islands both reduce the amount of JavaScript that has to hydrate.

Mix strategies within one product. For the invoicing app: static generation for the marketing site and help centre, client or server rendering for the logged-in dashboard, and server rendering with almost no JavaScript for the public "pay this invoice" page, because a client opening it from an email on a phone should see the amount and the Pay button immediately.

Core Web Vitals

Google's Core Web Vitals are three measurements of what users actually experience. Each is assessed at the 75th percentile of real visits, so three out of four visits must meet the threshold.

METRIC	MEASURES	GOOD	POOR	COMMON CAUSES OF A POOR SCORE
Largest Contentful Paint	Loading: when the main content appears	≤ 2.5s	> 4.0s	Slow server response, render-blocking resources, a heavy hero image, client-side rendering
Interaction to Next Paint	Responsiveness: from input to the next visual update, across the whole visit	≤ 200ms	> 500ms	Long JavaScript tasks, heavy hydration, expensive re-renders
Cumulative Layout Shift	Visual stability	≤ 0.1	> 0.25	Images without dimensions, late banners, fonts swapping in

Interaction to Next Paint replaced First Input Delay in March 2024, and it's the one most JavaScript-heavy apps struggle with. Lab tools such as Lighthouse, run on a fast developer laptop, are useful for diagnosis but can be wildly optimistic. What counts is field data from real users on real devices.

EXAMPLE vitals.js · measuring real users, with the web-vitals library

```
import { onCLS, onINP, onLCP } from "web-vitals";

function report(metric) {
  navigator.sendBeacon("/analytics/vitals", JSON.stringify({
    name: metric.name,           // "LCP", "INP" or "CLS"
    value: metric.value,
    rating: metric.rating,       // "good", "needs-improvement" or "poor"
    page: location.pathname,
  }));
}

onLCP(report);
onINP(report);
onCLS(report);
```

Bytes are a budget

JavaScript is the most expensive thing you can send. After downloading, it must be parsed, compiled and run on the main thread, and on a mid-range phone that takes several times longer than on your laptop. Set a budget, such as a maximum compressed size of JavaScript for each route, and fail the build when a change exceeds it. Then keep under it.

- **Split code by route**, so the dashboard doesn't download the settings page.
- **Load heavy, rarely used components on demand**: charts, rich-text editors, PDF previews.
- **Audit dependencies** with a bundle analyser. A date library imported for one format call can outweigh your whole application.
- **Prefer the platform**. `Intl.NumberFormat` formats money in every locale, and `<dialog>` is a modal, without a single byte of library code.

EXAMPLE the chart is downloaded only when it's shown

```
import { lazy, Suspense } from "react";

const RevenueChart = lazy(() => import("./RevenueChart"));

export function RevenuePanel({ monthly }: { monthly: MonthlyTotal[] }) {
  return (
    <Suspense fallback={<ChartSkeleton />}>
      <RevenueChart data={monthly} />
    </Suspense>
  );
}
```

Images and fonts

Images are usually the largest bytes on a page and often the Largest Contentful Paint element itself. Serve modern formats such as AVIF or WebP, offer several sizes with `srcset`, and always give `width` and `height` so the browser reserves space and nothing shifts. Lazy-load images below the fold, but never the main image at the top; mark that one `fetchpriority="high"` instead.

EXAMPLE

```

```

For fonts, use as few families and weights as you can, subset them to the characters you need, preload the main one, and use `font-display: swap` so text is readable before the font arrives.

Caching

Build tools put a content hash in each asset's file name, such as `app.3f9a1c.js`, which changes whenever the file does. That makes it safe to cache those files forever. The HTML that references them must never be cached like that, or users will keep loading old versions after every deploy.

EXAMPLE response headers

```
# app.3f9a1c.js: the name changes whenever the content does
Cache-Control: public, max-age=31536000, immutable

# index.html: always check with the server before reusing
Cache-Control: no-cache

# GET /api/invoices: personal data; never stored by shared caches
Cache-Control: private, no-store
```

A classic confusion: `no-cache` doesn't mean "don't cache". It means "check with the server before using the cached copy", which is exactly right for HTML. `no-store` is the one that means don't keep it at all.

Perceived performance

How fast something feels matters as much as how fast it is. Respond to every click within about 100 milliseconds, even if only by showing that something has started. Use skeletons that match the final layout. Prefetch the next likely page when the user hovers over a link. And use optimistic updates (chapter 10) for safe, reversible actions.

COMMON TRAPS

Testing only on a fast laptop. Your users' median device is slower than yours, often by several times.

Lazy-loading the hero image. It delays exactly the element that decides Largest Contentful Paint.

A whole library for one feature. Check the cost of every dependency.

Images without dimensions. The page jumps as they load.

Hydrating everything for one interactive widget.

Long-lived caching on HTML. Users are stuck on old versions after you deploy.

EXERCISES

EXERCISE 11.1 Choose the rendering

Choose a rendering strategy for each page of the invoicing app, with a reason.

- a. The marketing home page.
- b. The pricing page, which changes a few times a year.
- c. The logged-in dashboard.
- d. The public "pay this invoice" page that clients open from an email.
- e. A help centre with 400 articles.

SOLUTION

- a. **Static.** The same for everyone, and it must be fast for search engines and first impressions.
- b. **Static**, rebuilt when prices change. Currency can be localised by a small script or at the edge.
- c. **Client-side or server-side rendering.** It's personal and interactive. Server rendering with streaming shows the frame quickly while slower panels load.
- d. **Server-side, with almost no JavaScript.** It's personal, opened on phones from email, and every second of delay costs payments. Load the payment provider's script only when it's needed.
- e. **Static or incremental regeneration**, with a search index. Articles change occasionally, and there are too many to rebuild on every edit.

EXERCISE 11.2 Read the field report

Real-user data for the dashboard on mobile shows a 75th-percentile Largest Contentful Paint of 4.1 seconds, Interaction to Next Paint of 380 milliseconds, and layout shift of 0.02. A profile shows 1.2 MB of JavaScript, a 2.5 MB PNG in the header, and a 600-millisecond task each time the status filter changes. What do you fix, in what order?

SOLUTION

- 1. **The header image first.** Loading is in the poor band, above 4 seconds, and the image is probably the Largest Contentful Paint element. Converting it to AVIF at the right size, adding dimensions, and giving it high fetch priority is quick and high-impact.

2. **Then the JavaScript.** Split it by route and lazy-load the charts. That helps loading and hydration.
3. **Then the filter task.** Responsiveness is in the "needs improvement" band. Filter on the server, or memoise the filtering, render fewer rows with virtualisation, and yield to the browser during long work.
4. **Leave layout shift alone.** It's already good.

Then measure again with real users, because lab improvements don't always survive contact with real devices.

EXERCISE 11.3 Write the cache headers

Write a `Cache-Control` header for each response.

- a. A JavaScript bundle named `dashboard.8c21e7.js`.
- b. The HTML shell of the app.
- c. A user's invoice list from the API.
- d. A logo on the marketing site, named `logo.svg` without a hash.

SOLUTION

- a. `public, max-age=31536000, immutable`. The hash changes whenever the content does.
- b. `no-cache`, so the browser always checks for a newer version.
- c. `private, no-store`. Personal data must never sit in a shared cache.
- d. `public, max-age=86400, stale-while-revalidate=604800` is a reasonable compromise. Better still, have the build add a hash to the file name, then cache it forever like the bundle.

CHECKLIST

- ☐ Is the rendering strategy chosen page by page?
- ☐ Are Core Web Vitals measured from real users, at the 75th percentile?
- ☐ Is there a JavaScript budget, enforced in CI?
- ☐ Do images have dimensions, modern formats and the right loading priority?
- ☐ Are cache headers deliberate for assets, HTML and API responses?

GO DEEPER

GUIDE [Web Vitals web.dev](#)

GUIDE [Rendering on the Web web.dev](#)

COURSE [Learn Performance web.dev](#)

DOCS [HTTP caching MDN](#)

PART IV

Behind the request

Much of what makes a product work happens after the response is sent: the email that goes out, the PDF that renders, the file that uploads, the webhook that arrives at three in the morning, the payment that settles. Four chapters on that machinery.

Background jobs and scheduled work

3 exercises

Anything slow, unreliable, or not needed for the response belongs in a background job. Jobs run at least once, sometimes twice, and sometimes out of order, so make every job idempotent, retry with backoff, and give failures somewhere visible to land.

What belongs in a job

Sending email, rendering PDFs, calling third-party APIs, resizing images, imports and exports, delivering webhooks: anything that takes more than a few hundred milliseconds, or that can fail for reasons outside your control. The request records what needs to happen and returns quickly, often with 202 Accepted (chapter 1). A worker process does the work. The user learns the outcome from a status in the interface, a server-sent event, or an email.

A job system has three parts: producers that enqueue jobs, a **broker** that stores them, and **workers** that run them. The broker might be Redis, RabbitMQ or a cloud queue, or it can simply be your PostgreSQL database, which is a very good choice for a young product for reasons that will become clear below.

How queues actually deliver

A worker takes a job and starts it. If the worker crashes before confirming it's finished, the broker assumes the job was lost and, after a timeout, hands it to another worker. That's the only safe behaviour, and it means a job can run twice: the first run may have done most of its work before dying. This is **at-least-once delivery**, and it's what every practical queue provides. Exactly-once delivery isn't really on offer. What you can have is exactly-once *effects*, by making jobs idempotent. Don't count on order, either: retries and parallel workers mean jobs can complete in any sequence.

Idempotent jobs

BEFORE a crash between the two lines sends the email twice

```
@job
def send_invoice_email(invoice_id):
    invoice = db.get_invoice(invoice_id)
    mailer.send(invoice.client_email, render_invoice_email(invoice))
    db.execute("UPDATE invoices SET sent_at = now() WHERE id = %s", (invoice_id,))
```

AFTER safe to run again

```
@job(max_attempts=8, backoff="exponential")
def send_invoice_email(invoice_id):
    invoice = db.get_invoice(invoice_id)
    if invoice.sent_at is not None:
        return # an earlier run finished the job
    mailer.send(
        invoice.client_email,
        render_invoice_email(invoice),
        idempotency_key=f"invoice-email-{invoice_id}", # the provider drops repeats
    )
    db.execute(
        "UPDATE invoices SET sent_at = now() WHERE id = %s AND sent_at IS NULL",
        (invoice_id,))
```

There's still a gap between sending and recording. If your email provider accepts an idempotency key, pass one and the gap closes. If it doesn't, you must choose which way to fail. Record first and you risk never sending: at most once. Send first and you risk sending twice: at least once. For a receipt, an occasional duplicate is far better than a missing one. For a charge, neither gap is acceptable, which is why payment APIs support idempotency keys (chapter 15).

Pass identifiers in job payloads, not whole objects. The worker should load the current invoice when it runs, not act on a snapshot from minutes ago, and payloads containing personal data end up copied into logs and dashboards.

Retries, backoff and dead letters

Retry transient failures with exponential backoff and jitter, as *The Engineer's Craft* describes, and cap the number of attempts. Don't retry permanent failures at all: an email address the provider rejects as invalid will be just as invalid in an hour. When a job exhausts its attempts or fails permanently, move it to a **dead-letter queue** with the error attached, alert someone, and provide a way to inspect it and run it again once the cause is fixed.

The dual-write problem, and the outbox

This innocent code has two different ways to go wrong.

BEFORE two systems, two writes, no atomicity

```
def send_invoice(db, queue, invoice_id):
    db.execute("UPDATE invoices SET status = 'sent' WHERE id = %s", (invoice_id,))
    db.commit()
    queue.enqueue("send_invoice_email", invoice_id=invoice_id) # what if we die here?
```

If the process dies between the commit and the enqueue, the invoice says "sent" and no email ever goes. Swap the order and the job may run before the commit is visible, or the transaction may roll back after the job has been queued, emailing an invoice that was never

sent. The database and the queue are two systems, and there's no transaction spanning both.

The **transactional outbox** fixes this. Write the job, or event, into an `outbox` table in the *same* database transaction as the business change, so both commit or neither does. A separate relay process reads new outbox rows, puts them on the queue, and marks them published.

AFTER one transaction, then a relay

```
def send_invoice(db, invoice_id, user):
    with db.transaction():
        db.execute("UPDATE invoices SET status = 'sent' WHERE id = %s", (invoice_id,))
        db.execute(
            "INSERT INTO outbox (topic, payload) VALUES (%s, %s)",
            ("invoice.sent", json.dumps({"invoice_id": invoice_id, "sent_by": user.id})))

def relay_outbox(db, queue, batch_size=100):
    with db.transaction():
        rows = db.fetch_all(
            "SELECT id, topic, payload FROM outbox WHERE published_at IS NULL "
            "ORDER BY id LIMIT %s FOR UPDATE SKIP LOCKED", (batch_size,))
        for row in rows:
            queue.publish(row.topic, row.payload)          # at least once
            db.execute("UPDATE outbox SET published_at = now() WHERE id = %s", (row.id,))
```

`FOR UPDATE SKIP LOCKED` lets several relays run side by side, each claiming different rows. The relay can still publish a message and crash before marking it, so consumers see duplicates occasionally, and they must be idempotent. That's the point of the previous section. And if your job queue lives in the same PostgreSQL database, you get the same guarantee for free: enqueue the job inside the business transaction. That's a strong reason to start with a database-backed queue.

Scheduled work

Recurring work, such as "send reminders for overdue invoices every morning", brings three extra concerns. It must run **once**, not once per application server, so use a scheduler with unique jobs or take a database advisory lock. It must be **idempotent over its window**, so that running it twice on the same day sends each reminder once. And "every morning" raises the question of **whose** morning. Run the scheduler hourly and select the teams for whom it's now 9 a.m. locally.

EXAMPLE reminders.py

```
@scheduled(cron="0 * * * *")          # at the top of every hour, UTC
def queue_overdue_reminders(db, queue, now):
    for team in db.teams_where_local_hour_is(9, now):
        local_date = now.astimezone(team.timezone).date()
        for invoice in db.overdue_invoices_without_reminder_on(team.id, local_date):
            queue.enqueue(
```

```
        "send_overdue_reminder",
        invoice_id=invoice.id,
        unique_key=f"reminder:{invoice.id}:{local_date}",    # at most once a day
    )
```

Keep it visible

Queues fail quietly. Watch the depth of each queue, the failure rate, the size of the dead-letter queue, and above all the **age of the oldest waiting job**. A queue of 10,000 jobs moving quickly is healthy; a queue of 12 where the oldest has waited an hour is an incident. Break very long jobs into chunks, such as one job per thousand rows, so a crash doesn't restart from zero and a deploy doesn't have to wait an hour.

COMMON TRAPS

Slow work inside the request. The user waits, and a timeout loses the work.

Jobs that aren't idempotent. Duplicate emails, fees and exports.

Retrying permanent failures until the heat death of the universe.

Enqueueing outside the transaction. Lost jobs, or jobs for changes that never happened.

A scheduler on every server. Every reminder goes out once per machine.

Whole objects in payloads. Stale data, and personal data spread through your queue.

No alert on job age. The queue stalls on Friday night; customers notice on Monday.

EXERCISES

EXERCISE 12.1 Make the late fee idempotent

This job runs daily and occasionally twice. Make it apply the fee at most once per invoice per period, and announce the fee reliably.

```
@job
def apply_late_fee(invoice_id):
    invoice = db.get_invoice(invoice_id)
    invoice.amount_due_minor += LATE_FEE_MINOR
    db.save(invoice)
    queue.enqueue("email_late_fee_notice", invoice_id=invoice_id)
```

SOLUTION

```
@job
def apply_late_fee(invoice_id, period: str):          # for example "2026-09"
    with db.transaction():
        inserted = db.execute(
            "INSERT INTO late_fees (invoice_id, period, amount_minor) VALUES (%s, %s, %s) "
            "ON CONFLICT (invoice_id, period) DO NOTHING",
            (invoice_id, period, LATE_FEE_MINOR)).rowcount
        if inserted:
            db.execute(
                "INSERT INTO outbox (topic, payload) VALUES ('late_fee.applied', %s)",
                (json.dumps({"invoice_id": invoice_id, "period": period}),))
```

A unique constraint on invoice and period makes the second run a no-op, and the notice goes through the outbox in the same transaction. The fee is now a row rather than a change to a running total, so the amount due is calculated from line items plus fees, and it's obvious when and why each fee was added.

EXERCISE 12.2 Spot the dual write

This handler enqueues to Redis inside a database transaction. Describe what goes wrong, and when.

```
def cancel_subscription(db, redis_queue, team_id):
    with db.transaction():
        db.execute("UPDATE subscriptions SET status = 'cancelled' WHERE team_id = %s",
            (team_id,))
        redis_queue.enqueue("send_cancellation_email", team_id=team_id)
        db.execute("INSERT INTO audit_log (team_id, action) VALUES (%s, 'cancel')",
            (team_id,))
```

SOLUTION

- If the audit insert fails, the transaction rolls back and the subscription stays active, but the job is already in Redis. The customer receives a cancellation email for a subscription that's still running, and still billing.
- Even when everything succeeds, a fast worker can pick up the job before the commit is visible, read the subscription, and see it still active.

Fix it by writing the job to an outbox table inside the transaction and relaying it afterwards, or by using a queue that lives in the same database.

EXERCISE 12.3 Design the reminder schedule

Reminders should go out at 9 a.m. in the team's time zone on days 1, 7 and 14 after the due date, never at weekends, and at most three per invoice. Describe the scheduled job, the query it needs, and the key that prevents duplicates.

SOLUTION

- An hourly job selects teams for whom it's between 9:00 and 9:59 local time on a weekday.
- For each team, select unpaid invoices where the days since the due date, in the team's local calendar, are 1, 7 or 14, and where fewer than three reminders have been sent.
- Decide what happens when a reminder day falls at a weekend: send on the following Monday, for instance, which means selecting "days overdue $\geq N$, and reminder N not yet sent" rather than "exactly N ".
- Enqueue with the unique key `reminder:{invoice_id}:{n}`, where `n` is 1, 2 or 3, so no reminder can be sent twice.
- Record each sent reminder in a `reminders` table with the same unique key, which also gives support a history to look at.

CHECKLIST

- ☐ Is every slow or unreliable side effect in a job rather than the request?
- ☐ Is every job idempotent, with payloads that carry identifiers rather than data?
- ☐ Do retries back off, stop on permanent errors, and end in a visible dead-letter queue?
- ☐ Are changes and their jobs written atomically, through an outbox or a same-database queue?
- ☐ Do you alert on the age of the oldest waiting job?

GO DEEPER

PATTERN Transactional outbox [Chris Richardson, microservices.io](https://microservices.io)

GUIDE Transactional outbox pattern [AWS Prescriptive Guidance](#)

ESSAY Transactionally Staged Job Drains in Postgres [Brandur Leach](#)

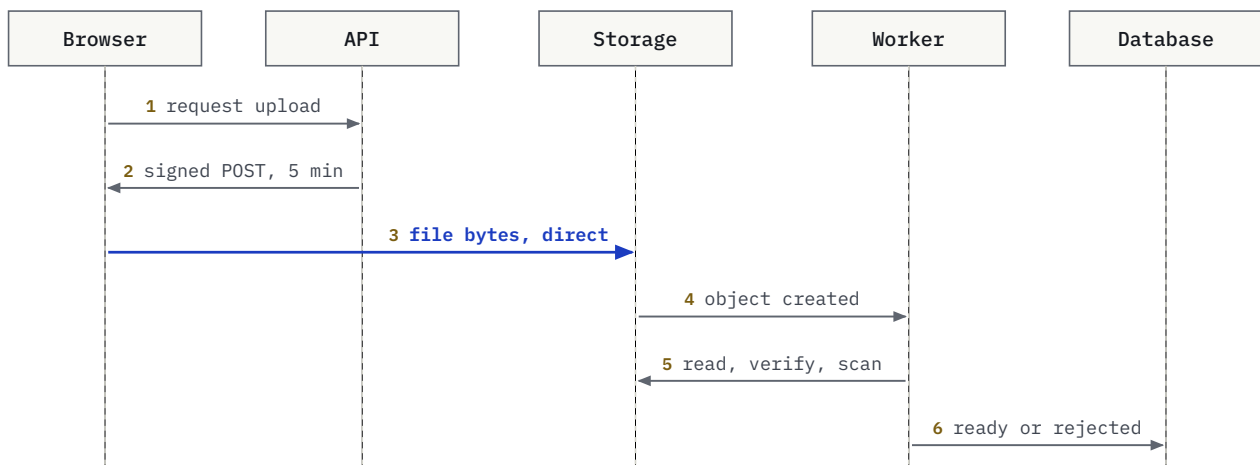
Files, uploads and CDNs

3 exercises

Files are untrusted input with a size. Upload them straight to object storage with a narrow, short-lived permission, never trust their names or declared types, check them in the background, and serve them with the right access rules.

Keep file bytes away from your app servers

Streaming a 40 MB upload through your API ties up a worker for the whole transfer, uses memory, and runs into request timeouts on slow connections. Instead, let the browser upload directly to object storage, such as S3 or any compatible service. Your API's only job is to decide whether the upload is allowed and to sign a permission for exactly that upload.



The file travels straight from the browser to storage. Your API only signs a narrow, short-lived permission, and a background worker checks the file before anyone else can download it.

EXAMPLE uploads.py, with boto3

```

import uuid
import boto3

s3 = boto3.client("s3")
ALLOWED_TYPES = {"image/png": "png", "image/jpeg": "jpg", "application/pdf": "pdf"}
MAX_BYTES = 10 * 1024 * 1024

def create_upload(team_id: str, content_type: str) -> dict:
    if content_type not in ALLOWED_TYPES:
        raise UnsupportedFileType(content_type)
    key = f"uploads/{team_id}/{uuid.uuid4()}.{ALLOWED_TYPES[content_type]}" # we choose it
    return s3.generate_presigned_post(

```

```

    Bucket="invoice-app-uploads",
    Key=key,
    Fields={"Content-Type": content_type},
    Conditions=[
        {"Content-Type": content_type},
        ["content-length-range", 1, MAX_BYTES],      # storage rejects anything larger
    ],
    ExpiresIn=300,
)

```

The signed policy is enforced by the storage service itself: the exact key, the declared type, a size range, and a five-minute lifetime. The browser posts the file with the returned fields, then tells your API the upload is finished, or storage sends an event to a worker directly.

Never trust the file

- **The name is hostile.** User-supplied file names can contain `../` path traversal, control characters, or markup that breaks your pages. Generate your own storage keys, as above. Keep the original name as metadata, cleaned up, only for display.
- **The type is a claim.** A file's extension and its declared content type are whatever the uploader said. Check the actual bytes after upload: a PDF starts with `%PDF-`, a PNG with an eight-byte signature, a JPEG with `FF D8 FF`.
- **Files other people will open need scanning.** Run a malware scanner in the processing job and quarantine files until they pass.
- **Re-encode images.** Decoding and re-saving an image removes metadata, such as GPS coordinates in a phone photo, and neutralises many malformed-file exploits. Cap pixel dimensions first, to avoid decompression bombs that expand to gigabytes.
- **SVG is code.** An SVG file is XML that can contain scripts. Don't serve users' SVGs from your own domain unless you've sanitised them; rasterising them is safer.

EXAMPLE checking the real type

```

SIGNATURES = {
    "image/png": b"\x89PNG\r\n\x1a\n",
    "image/jpeg": b"\xff\xd8\xff",
    "application/pdf": b"%PDF-",
}

def looks_like(declared_type: str, first_bytes: bytes) -> bool:
    signature = SIGNATURES.get(declared_type)
    return signature is not None and first_bytes.startswith(signature)

```

A processing pipeline

Give every upload a status the interface can show: `pending` while it's uploading, `processing` while a job checks the type, scans it and generates thumbnails, then `ready` or `rejected` with a reason a person can act on, such as "This file isn't a PDF". The job follows chapter 12: idempotent, retried on transient failures, and failing into a visible queue.

Serving files

Public files, such as product images and your own static assets, go through a CDN with content-hashed names and long cache lifetimes (chapter 11). **Private files**, such as invoice PDFs and receipts, stay in a private bucket. When a user asks for one, check authorisation (chapter 7), then redirect to a signed URL that expires within a minute or two.

EXAMPLE a private download

```
def invoice_pdf_url(user, invoice) -> str:
    authorize(user, "invoice:read", invoice)
    return s3.generate_presigned_url(
        "get_object",
        Params={
            "Bucket": "invoice-app-private",
            "Key": invoice.pdf_key,
            "ResponseContentDisposition": f'attachment; filename="{invoice.number}.pdf"',
        },
        ExpiresIn=60,
    )
```

Serve user-uploaded content from a separate domain, such as `files.example-usercontent.com`, so that even a file that slips through your checks can't run script with access to your application's cookies. Send `X-Content-Type-Options: nosniff` so browsers don't second-guess the type, and `Content-Disposition: attachment` for anything that should download rather than open.

Don't email signed storage URLs. They either expire before the recipient clicks, or live long enough to be forwarded, leaked and reused. Email a link to a page in your app, which checks access and then issues a fresh, short-lived URL.

Deleting and keeping

When a record is deleted, delete its files too. Orphaned files cost money, and under privacy laws such as the GDPR a user's right to erasure includes their files. Set lifecycle rules so abandoned uploads, which were signed but never completed, disappear after a day. Decide deliberately how long backups and object versions keep deleted files, and say so in your privacy policy.

COMMON TRAPS

The user's file name as the storage key. Path traversal, collisions and overwrites.

Trusting the declared type. A script renamed to `invoice.pdf` is still a script.

Public buckets for private documents. One guessed or leaked URL exposes them for good.

Serving uploads from your app's own domain. A hostile file runs with your cookies.

Long-lived signed URLs in emails. Forwarded, indexed, and leaked.

Location data left in photos. A user's home address, shared with everyone who sees the image.

Orphaned files after records are deleted.

EXERCISES

EXERCISE 13.1 Review the upload handler

List every problem with this receipt-upload endpoint, and describe the fix.

```
@app.post("/receipts")
def upload_receipt(file: UploadFile):
    if not file.filename.endswith(".pdf"):
        raise BadRequest("PDF only")
    path = f"/var/app/uploads/{file.filename}"
    with open(path, "wb") as out:
        out.write(file.file.read())
    return {"url": f"https://app.example.com/uploads/{file.filename}"}
```

SOLUTION

- **No authorisation or tenant scoping.** Anyone can upload, and files aren't tied to a team.
- **The user's name is the path.** `../../app/config.pdf` escapes the directory, and two users with the same file name overwrite each other.
- **The extension is the only check.** The bytes could be anything.
- **No size limit,** and the whole file is read into memory.
- **Stored on the app server's disk,** which is lost on redeploy and not shared between servers.
- **Served publicly from the app's own domain.** No access control, and hostile content runs under your origin.

The fix is this chapter: a signed direct upload with a generated key and size limit, a job that checks the bytes and scans the file, a private bucket, and downloads through an authorised, short-lived signed URL from a separate domain.

EXERCISE 13.2 Check the bytes

Four files were uploaded as logos, all declared as `image/png`. Here are their first bytes. Which do you accept?

```
A: 89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52
B: 25 50 44 46 2D 31 2E 37 0A 25 E2 E3 CF D3 0A 31
C: 3C 73 76 67 20 78 6D 6C 6E 73 3D 22 68 74 74 70
D: FF D8 FF E0 00 10 4A 46 49 46 00 01 01 00 00 01
```

SOLUTION

- A is a real PNG: the signature `89 50 4E 47 0D 0A 1A 0A` spells `"\x89PNG\r\n\x1a\n"`. Accept it, then re-encode it.
- B is a PDF: the bytes spell `%PDF-1.7`. Reject it.
- C is an SVG: the bytes spell `<svg xmlns="http`. Reject it, or rasterise it in a sandbox. Never serve it as-is from your domain.
- D is a JPEG, not a PNG. It's harmless, so you might accept it and re-encode, but the declared type was wrong, which is worth logging.

EXERCISE 13.3 Share a private PDF

A freelancer emails an invoice to a client who has no account. The link must work for 30 days, but a forwarded or leaked link shouldn't give permanent access, and the freelancer should be able to revoke it. Design it.

SOLUTION

- Email a link to an app page, `/i/{token}`, where the token is long and random, stored hashed, scoped to one invoice, and expires in 30 days.
- The page shows the invoice and a Download button. Each download checks the token, then redirects to a signed storage URL valid for 60 seconds.
- The freelancer can revoke or regenerate the link from the invoice page, which deletes the token.

- Log each view, with a time and a rough location, so the freelancer can see "viewed by client" and spot something odd.

CHECKLIST

- ☐ Do uploads go directly to storage, with a short, narrow signature and a size limit?
- ☐ Do you choose the storage keys, and check files by their bytes rather than their names?
- ☐ Are files scanned and re-encoded before other people can download them?
- ☐ Are private files served only through short-lived signed URLs, after an authorisation check?
- ☐ Are uploads served from a separate domain, and deleted along with their records?

GO DEEPER

GUIDE [File Upload Cheat Sheet OWASP](#)

DOCS [Working with presigned URLs Amazon S3](#)

DOCS [generate_presigned_post boto3 reference](#)

Webhooks, integrations and email

3 exercises

Every integration is a distributed system you don't control. Verify what arrives, acknowledge it quickly, process it idempotently, and reconcile what you missed. When you send email, prove to the inbox that it really comes from you.

Receiving webhooks

A webhook is an HTTP request another service sends you when something happens: a payment succeeded, an email bounced, a subscription renewed. Your endpoint is on the public internet, so anyone can send it anything, and the sender will retry, duplicate and reorder deliveries. Seven rules cover it.

- 01 **Verify the signature** over the raw request body, with the shared secret, using a constant-time comparison.
- 02 **Check the timestamp** is recent, typically within five minutes, so a captured request can't be replayed later.
- 03 **Deduplicate by event id.** Senders retry whenever they're unsure you received something, so the same event can arrive twice.
- 04 **Acknowledge fast.** Store the event and return 2xx within a few seconds, then do the real work in a background job.
- 05 **Don't trust the payload for anything important.** Fetch the current state of the object from the provider's API.
- 06 **Expect any order.** "Updated" can arrive before "created". Compare timestamps or versions, or simply fetch the current state.
- 07 **Reconcile.** Webhooks get lost, and a sender disables endpoints that keep failing; Stripe does so after three days of failed retries. A daily job that lists recent objects from the provider and fixes anything you missed turns a silent loss into a delay.

Many providers now follow the Standard Webhooks specification: three headers carry an event id, a Unix timestamp and a signature, and the signature is an HMAC-SHA256 of the id, the timestamp and the body joined by dots. During a secret rotation, the signature header can carry several signatures, and any one matching is enough.

EXAMPLE webhooks.py · verification

```
import base64
import hashlib
```

```

import hmac
import time

TOLERANCE_SECONDS = 300

def verify_webhook(secret: bytes, headers: dict, raw_body: bytes) -> None:
    msg_id = headers["webhook-id"]
    timestamp = headers["webhook-timestamp"]
    if abs(time.time() - int(timestamp)) > TOLERANCE_SECONDS:
        raise InvalidSignature("timestamp outside tolerance")

    signed = f"{msg_id}.{timestamp}".encode() + raw_body
    expected = base64.b64encode(hmac.new(secret, signed, hashlib.sha256).digest()).decode()

    # During secret rotation the header carries several: "v1,<sig> v1,<sig>"
    for candidate in headers["webhook-signature"].split():
        version, _, signature = candidate.partition(",")
        if version == "v1" and hmac.compare_digest(signature, expected):
            return
    raise InvalidSignature("no signature matched")

```

EXAMPLE webhooks.py · the endpoint

```

@app.post("/webhooks/payments")
async def payments_webhook(request: Request):
    raw = await request.body() # the exact bytes that were signed
    verify_webhook(WEBHOOK_SECRET, request.headers, raw)
    event = json.loads(raw)
    inserted = db.execute(
        "INSERT INTO webhook_events (id, type, payload) VALUES (%s, %s, %s) "
        "ON CONFLICT (id) DO NOTHING",
        (event["id"], event["type"], raw)).rowcount
    if inserted:
        queue.enqueue("process_webhook_event", event_id=event["id"])
    return Response(status_code=200) # acknowledge; the job does the work

```

Verify against the raw bytes, before any parsing. Many frameworks parse JSON automatically, and if you re-serialise the parsed object to check the signature, spacing and key order change, and every valid webhook fails verification.

Sending webhooks

If your own product sends webhooks to customers, you're now the sender, and the same rules apply in reverse. Sign every delivery. Give every event a unique id and a type. Retry failures with exponential backoff over a day or more, then disable the endpoint and tell the customer. Show customers a log of recent deliveries with a button to resend. Support secret rotation. And keep payloads small; many APIs send only the event type and object id, so the receiver fetches the current object and never acts on stale data.

One risk is specific to senders: customers choose the destination URL. Without checks, someone registers an internal address and your servers helpfully post to your own private network. Validate destinations as chapter 16 describes.

Calling third-party APIs

Everything from *The Engineer's Craft* applies. Put each provider behind an adapter you own. Set a timeout on every call. Retry only transient failures of idempotent operations, with backoff and jitter. Respect the provider's rate limits and its `Retry-After` header. Use its test mode or sandbox in development. Log enough about each call to debug it later, without logging secrets. And decide in advance what your product does while the provider is down: queue the work and retry, show a clear message, or fall back to a manual path.

Transactional email

Password resets, receipts, invoices and reminders are transactional email: expected, individual, and important. Send it through a dedicated email provider, never directly from your servers, and keep it separate from marketing email, on its own sending stream or subdomain. If a newsletter draws spam complaints, it mustn't drag your password resets into the spam folder with it.

Inbox providers need proof that mail claiming to be from your domain really is. Three DNS records provide it.

- **SPF** lists the servers allowed to send mail for your domain.
- **DKIM** signs each message cryptographically, with a public key published in DNS.
- **DMARC** tells receivers what to do with mail that fails those checks, and where to send reports.

EXAMPLE DNS records, simplified

```
example.com.          TXT "v=spf1 include:spf.mailprovider.example -all"
pm2026._domainkey.example.com. TXT "v=DKIM1; k=rsa; p=MIIBIjANBgkqhkiG9w0BAQEF..."
_dmarc.example.com.   TXT "v=DMARC1; p=quarantine; rua=mailto:dmarc-
reports@example.com"
```

Since February 2024, Gmail and Yahoo have required these from anyone sending 5,000 or more messages a day to their users, along with one-click unsubscribe headers (RFC 8058) on marketing mail, and a spam-complaint rate kept below 0.3%, and ideally below 0.1%. Gmail moved to rejecting non-compliant mail in November 2025. Set them up before launch, even while you're small; retrofitting them after deliverability has collapsed is much harder.

Handle the feedback. Providers report bounces and complaints, usually by webhook. Stop sending to addresses that hard-bounce, honour every unsubscribe immediately, and watch your complaint rate. In the messages themselves, include a plain-text version, use real text rather than text baked into images, and link to pages in your app rather than to raw file URLs.

Never let a development or staging environment send email to real people. Use the provider's sandbox mode, or route every outgoing message to a catch-all inbox. Chapter 20 comes back to this, because it's one of the most common and most embarrassing launch mistakes.

Notifications in general

Let users choose which notifications they get, on which channel, per type. Batch noisy events into digests. Respect quiet hours and time zones. And don't notify people about their own actions: a freelancer who just paid their subscription doesn't need an email telling them a payment was made.

COMMON TRAPS

Parsing before verifying. The signature covers raw bytes, not your re-serialised JSON.

Doing the work inside the webhook request. Slow responses cause retries, and retries cause duplicates.

No deduplication. Every retried delivery is processed again.

Trusting delivery order. The older event arrives last and overwrites the newer state.

No reconciliation. A lost webhook means a customer who paid but is marked unpaid, forever.

Marketing and transactional mail mixed together. A newsletter complaint breaks password resets.

Real email from staging. Your test invoices, delivered to real customers.

EXERCISES

EXERCISE 14.1 Review the handler

List every problem with this payment webhook handler.

```
@app.post("/webhooks/payments")
def payments_webhook(event: dict):
    if event["type"] == "payment.succeeded":
        invoice = db.find_invoice_by_amount(event["data"]["amount"])
        invoice.status = "paid"
        db.save(invoice)
        mailer.send(invoice.owner_email, "You got paid!")
    return {"ok": True}
```

SOLUTION

1. **No signature check.** Anyone can mark any invoice paid by posting JSON.
2. **The body is already parsed,** so verifying the raw bytes is no longer possible.
3. **Matched by amount.** Two invoices for €500 are indistinguishable. Match by the payment's own id or the invoice id in its metadata.
4. **No deduplication.** A retried delivery sends a second "You got paid!" email.
5. **Work done inline.** A slow email provider delays the response past the sender's timeout, which triggers a retry.
6. **No out-of-order handling.** A late "succeeded" could overwrite a later "refunded".
7. **Any exception becomes a 500,** so the sender retries events that were half processed.

Rewrite it as in this chapter: verify raw bytes, store the event with `ON CONFLICT DO NOTHING`, return 200, and process in a job that fetches the current payment from the provider and updates the invoice idempotently.

EXERCISE 14.2 Events out of order

You receive two `subscription.updated` events for the same team. The first to arrive says the plan is "pro" and was created at 10:00:05. The second to arrive says "basic" and was created at 10:00:02. What should you store, and how do you make this safe in general?

SOLUTION

Store "pro". The "basic" event is older; it just arrived later.

In general, either keep the timestamp or version of the last event applied and ignore anything older, or, more robustly, treat every event as a hint: when one arrives, fetch the subscription's current state from the provider's API and store that. Then order stops mattering, and a missed event is corrected by the next one.

EXERCISE 14.3 Diagnose the spam folder

Password-reset emails are landing in spam at Gmail. You find that your SPF record lists only your previous email provider, the new provider isn't signing with DKIM for your domain, there's no DMARC record, and your weekly newsletter goes out from the same address with a 0.5% complaint rate. What do you fix, in what order?

SOLUTION

1. **Authentication first.** Add the new provider to SPF and remove the old one. Set up DKIM signing for your domain at the new provider.
2. **Publish DMARC**, starting with `p=none` and a reporting address, so you can see who sends mail as you. Move to `quarantine` once the reports are clean.
3. **Separate the streams.** Send the newsletter from a marketing subdomain or stream, with one-click unsubscribe headers.
4. **Cut the complaints.** 0.5% is above the 0.3% ceiling. Send newsletters only to people who opted in, remove inactive addresses, and make unsubscribing easy.
5. **Monitor** with Google's Postmaster Tools and your DMARC reports.

CHECKLIST

- ☐ Are webhook signatures verified over the raw bytes, with a timestamp tolerance?
- ☐ Are events deduplicated, acknowledged quickly, processed in jobs, and reconciled daily?
- ☐ Do webhooks you send carry signatures and ids, retry, and support replay and secret rotation?
- ☐ Is every third-party call behind an adapter, with timeouts and sensible retries?
- ☐ Is email authenticated with SPF, DKIM and DMARC, with transactional mail kept separate and bounces handled?

GO DEEPER

SPEC [Standard Webhooks the specification](#)

DOCS [Receive events in your webhook endpoint Stripe](#)

GUIDE [Email sender guidelines Google](#)

Payments

3 exercises

Payments are where a bug costs real money the same day. Let a payment provider handle the card, never trust a price sent by the browser, treat the provider's webhooks as the source of truth, and make every call that moves money idempotent.

Don't touch card numbers

The Payment Card Industry Data Security Standard, PCI DSS, governs anyone who handles card data, and handling raw card numbers brings hundreds of requirements. Use your provider's hosted checkout page, or its embedded card fields, which send card details from the browser straight to the provider without passing through your servers. That puts most merchants in the lightest self-assessment category, called SAQ A.

It doesn't make you exempt. Since version 4.0.1 of the standard became fully mandatory in March 2025, SAQ A merchants must also confirm that their payment page isn't open to malicious scripts, because a script on your checkout page can read what the customer types even into a provider's fields. Following the provider's integration instructions covers much of this. The rest is on you: keep third-party scripts off pages that take payment.

The shape of a payment

Modern card payments are a small state machine, not a single call, because the customer's bank may ask them to confirm, usually through 3-D Secure. In Europe, Strong Customer Authentication rules make that step routine.

- 01 Your server creates a payment for an amount *it* calculated, and gets back a client secret.
- 02 The browser uses the provider's library and that secret to collect card details and confirm the payment, including any bank challenge.
- 03 The provider tells your server the outcome by webhook: succeeded, failed, or still processing.
- 04 Your server fulfils the order, here by marking the invoice paid, in response to that webhook.

EXAMPLE payments.py · step 1, with Stripe's Python library

```
import stripe
```

```
def start_invoice_payment(invoice, idempotency_key: str) -> str:
    amount = invoice_total_minor(invoice)          # from our records, never from the client
    intent = stripe.PaymentIntent.create(
        amount=amount,
        currency=invoice.currency.lower(),
        metadata={"invoice_id": invoice.id, "team_id": invoice.team_id},
        automatic_payment_methods={"enabled": True},
        idempotency_key=idempotency_key,           # a retry can't create a second payment
    )
    db.execute("UPDATE invoices SET payment_intent_id = %s WHERE id = %s",
               (intent.id, invoice.id))
    return intent.client_secret                    # the browser confirms the payment with this
```

EXAMPLE payments.py · step 4, the job from chapter 14

```
def process_webhook_event(event_id: str) -> None:
    event = stripe.Event.retrieve(event_id)         # fetch it from the provider; don't trust a
                                                    # copy
    if event.type != "payment_intent.succeeded":
        return
    intent = event.data.object
    with db.transaction():
        updated = db.execute(
            "UPDATE invoices SET status = 'paid', paid_at = now() "
            "WHERE id = %s AND payment_intent_id = %s AND status <> 'paid'",
            (intent.metadata["invoice_id"], intent.id)).rowcount
        if updated:                                # only the first delivery does anything
            db.execute(
                "INSERT INTO outbox (topic, payload) VALUES ('invoice.paid', %s)",
                (json.dumps({"invoice_id": intent.metadata["invoice_id"]}),))
```

Why fulfil from the webhook rather than from the browser returning to your "thank you" page? Because users close tabs, phones lose signal after paying, and a redirect URL can be visited by anyone. The webhook, verified and re-fetched, is the only reliable record that money moved.

Never trust the client with prices

The browser should send identifiers, such as which invoice or which plan, and never amounts, currencies, quantities or discounts. Anything in a request can be edited. If your server takes `{"amount": 49.00}` from the client, someone will send `{"amount": 0.49}`.

Money in code

Store money as an integer number of minor units with its currency, or as a decimal, never as a float (*The Engineer's Craft* explains why). The float bug is real and quiet: in Python, `int(19.99 * 100)` is `1998`, because 19.99×100 is actually `1998.9999999999998`. And "minor units" aren't always cents. Japanese yen has no minor unit, so 5,000 yen is sent as `5000`, while the Kuwaiti dinar has three decimal places. Keep a table of each currency's exponent from the ISO 4217 standard, and convert with decimals.

EXAMPLE money.py

```
from decimal import Decimal, ROUND_HALF_UP

MINOR_UNIT_DIGITS = {"USD": 2, "EUR": 2, "GBP": 2, "JPY": 0, "KWD": 3}

def to_minor_units(amount: str, currency: str) -> int:
    exponent = MINOR_UNIT_DIGITS[currency]
    scaled = Decimal(amount) * (10 ** exponent)
    return int(scaled.to_integral_value(rounding=ROUND_HALF_UP))

# to_minor_units("19.99", "USD") == 1999, to_minor_units("5000", "JPY") == 5000
```

Decide, and write down, when rounding happens and how: per line or on the total, half-up or half-even. Tax across many jurisdictions is a specialism of its own; use a tax calculation service rather than a table you maintain by hand.

Subscriptions

If the invoicing app charges freelancers monthly, you've entered the world of billing cycles, prorated plan changes, free trials, failed renewals and cancellations. Let your provider's billing product handle the mechanics, and keep a local copy of each subscription's state, updated by webhooks. Derive what a team is allowed to do, its **entitlements**, from that local copy; don't call the provider on every page load. Plan for **dunning**, the process of recovering failed renewals: retry the card on a schedule, email the customer, allow a grace period, and only then restrict the account.

Refunds, disputes and reconciliation

A refund is a new transaction, never a deletion; record who issued it and why. A **dispute**, or chargeback, is when the cardholder asks their bank to reverse a payment. It has deadlines, requires evidence, and costs a fee even if you win. And every day, **reconcile**: compare your record of payments, refunds and fees against the provider's balance transactions and payouts, and investigate every difference. Record money movements in an append-only ledger table; for anything beyond simple invoicing, learn double-entry bookkeeping, which accountants have been using to catch exactly these errors for five hundred years.

Testing payments

Every provider has a test mode with special card numbers that succeed, decline, or trigger a bank challenge; Stripe's classic success card is `4242 4242 4242 4242`. Use its command-line tool to forward test webhooks to your laptop. Then test the unhappy paths deliberately, because that's where the money leaks: a declined card, an abandoned bank challenge, a webhook delivered late, the same webhook delivered twice, a refund, and a dispute.

COMMON TRAPS

Amounts from the browser. Anything the client sends can be edited.

Fulfilling on the redirect. Tabs close, signals drop, and URLs can be typed.

Calls without idempotency keys. A retried timeout becomes a double charge.

Floats for money, and assuming every currency has two decimal places.

Deleting payment records. Refunds are new rows, not deletions.

Asking the provider on every request whether a team has paid.

Third-party scripts on the checkout page. They can read what customers type.

EXERCISES

EXERCISE 15.1 Find the money bugs

Find every problem in this payment endpoint.

```
@app.post("/pay")
def pay(body: dict, user=Depends(current_user)):
    invoice = db.get_invoice(body["invoice_id"])
    for attempt in range(3):
        try:
            charge = provider.charges.create(amount=int(body["amount"] * 100),
                                              currency="usd", source=body["card_token"])
            break
        except provider.Timeout:
            continue
    invoice.status = "paid"
    db.save(invoice)
    return {"ok": True}
```

SOLUTION

1. **The amount comes from the client.** Compute it from the invoice.
2. **Float arithmetic.** `int(19.99 * 100)` charges 1998 cents.
3. **The currency is hard-coded,** whatever the invoice says.
4. **Retries without an idempotency key.** A timeout after the charge succeeded means charging two or three times.
5. **Marked paid unconditionally.** If all three attempts time out, the loop ends and the invoice is still marked paid.
6. **No authorisation.** Any user can pay, or mark paid, any invoice id.

7. **Fulfilment in the request** rather than from the verified webhook, and no handling of bank challenges.

EXERCISE 15.2 Design the invoice states

Design the states of an invoice from the moment it's sent, and the payment events that move it between them. Include partial payments, refunds and disputes.

SOLUTION

FROM	EVENT	TO
sent	payment succeeded, less than the amount due	partially_paid
sent, partially_paid	payments reach the amount due	paid
sent, partially_paid	due date passes	overdue, with the paid amount kept
overdue	payments reach the amount due	paid
paid	full refund issued	refunded
paid	partial refund issued	paid, with a refund recorded
paid	dispute opened	disputed
disputed	dispute won or lost	paid, or refunded

Status is derived from the payments, refunds and disputes recorded against the invoice, never set directly. That keeps it correct when events arrive twice or out of order, and gives you a full history for free.

EXERCISE 15.3 Convert the amounts

Convert each amount to the integer a payment provider expects: 49.99 USD, 5,000 JPY, 12.345 KWD and 0.10 EUR. Then explain why `int(amount * 100)` is the wrong formula even for dollars.

SOLUTION

4999, 5000, 12345 and 10.

Multiplying by 100 assumes two decimal places, which is wrong for yen and dinars. And float multiplication truncated by `int()` is wrong even for dollars: 19.99×100 in floating point is 1998.9999999999998, which truncates to 1998. Parse the amount as a `Decimal`, scale by the currency's exponent, and round explicitly.

CHECKLIST

- ☐ Do card details go straight from the browser to the provider, with few scripts on the payment page?
- ☐ Are amounts calculated on the server from your own records?
- ☐ Is fulfilment driven by verified webhooks, idempotently?
- ☐ Does every call that creates, captures or refunds carry an idempotency key?
- ☐ Is money an integer or decimal with its currency, respecting each currency's minor units?
- ☐ Do you reconcile against the provider every day?

GO DEEPER

DOCS [Accept a payment Stripe](#)

DOCS [Testing Stripe's test cards and scenarios](#)

GUIDE [New SAQ A eligibility criteria PCI Security Standards Council](#)

ESSAY [Accounting for Developers Modern Treasury](#)

PART V

Protecting and releasing

Two chapters on the edges of a product: the abuse that arrives as soon as anything is public, and the discipline of shipping changes to real users without betting the company on every deploy.

Abuse, rate limits and input validation

3 exercises

Anything public will be abused: scraped, brute-forced, spammed and fed inputs you never imagined. Limit rates, put a bound on every resource, validate at the edge, and never fetch a URL a user gave you without checking where it leads.

Rate limiting

Rate limits stop password guessing, scraping, runaway client bugs and single tenants starving everyone else, and they cap what an attacker can cost you. The OWASP API Security Top 10 lists the lack of them as **unrestricted resource consumption**.

ALGORITHM	HOW IT WORKS	TRADE-OFF
Fixed window	Count requests per clock minute	Simple, but allows double bursts across a window boundary
Sliding window	Count requests in the last 60 seconds	Accurate; the exact version stores a timestamp per request
Token bucket	Tokens refill at a steady rate up to a maximum; each request spends one	Allows short bursts but enforces an average; the most common choice

EXAMPLE limits.py · a token bucket

```
import time
from dataclasses import dataclass

@dataclass
class Bucket:
    tokens: float
    updated: float

class TokenBucketLimiter:
    """Allow bursts of `burst` requests per key, refilling at `rate_per_sec`.

    Kept in memory here for clarity. Across several servers, keep buckets in a
    shared store such as Redis and update them atomically, for example in a script.
    """

    def __init__(self, rate_per_sec: float, burst: int, clock=time.monotonic):
        self.rate, self.burst, self.clock = rate_per_sec, burst, clock
        self.buckets: dict[str, Bucket] = {}

    def allow(self, key: str) -> tuple[bool, float]:
```

```

"""Return (allowed, seconds until the next request would be allowed)."""
now = self.clock()
bucket = self.buckets.get(key) or Bucket(tokens=self.burst, updated=now)
bucket.tokens = min(self.burst, bucket.tokens + (now - bucket.updated) * self.rate)
bucket.updated = now
self.buckets[key] = bucket
if bucket.tokens >= 1:
    bucket.tokens -= 1
    return True, 0.0
return False, (1 - bucket.tokens) / self.rate

```

This is the deep `RateLimiter` from *The Engineer's Craft*, made concrete: one method, `allow`, hides the algorithm, the clock and the storage. When a request is refused, return `429 Too Many Requests` with a `Retry-After` header computed from the second value, so well-behaved clients know exactly how long to wait.

Choose keys to match the threat. Limit sign-in attempts per account and per IP address, because attackers rotate both. Limit API calls per key and per tenant. Be careful with IP-only limits: an office or a mobile carrier can put thousands of people behind one address. Put coarse limits at the edge, in your CDN or gateway, to absorb floods cheaply, and business-specific limits in the application, where you know who the user is.

Put a bound on everything

Every unbounded input is a way to make your servers do unbounded work. Set a maximum request body size, a maximum page size (chapter 2), a maximum number of items in a batch, a maximum upload size (chapter 13), a maximum length for every string, a timeout on every operation, and per-tenant quotas on expensive actions such as exports. Tie quotas to plans, so the limits are also part of your pricing.

Regular expressions deserve a special mention. A pattern with nested repetition, such as `^([a-z0-9]+)*@`, can take exponential time on a crafted input like a long string of letters followed by an unexpected character. This is called ReDoS, regular expression denial of service. Limit the input's length before matching, and avoid nested quantifiers.

Validate at the edge, with allowlists

Parse every request into a strict, typed schema at the boundary, as in *The Engineer's Craft's* "parse, don't validate". Describe what is allowed, not what's forbidden: an allowlist of expected values is finite, while a blocklist of dangerous ones never is.

EXAMPLE `schemas.py`, with Pydantic

```

from datetime import date
from decimal import Decimal
from typing import Literal

from pydantic import BaseModel, ConfigDict, Field, model_validator

class LineIn(BaseModel):

```

```

model_config = ConfigDict(extra="forbid")
description: str = Field(min_length=1, max_length=500)
quantity: int = Field(ge=1, le=10_000)
unit_price: Decimal = Field(gt=0, max_digits=12, decimal_places=2)

```

```

class InvoiceIn(BaseModel):
    model_config = ConfigDict(extra="forbid")
    client_id: str = Field(pattern=r"^\d{12}$")
    currency: Literal["USD", "EUR", "GBP"]
    issue_date: date
    due_date: date
    lines: list[LineIn] = Field(min_length=1, max_length=200)

    @model_validator(mode="after")
    def due_on_or_after_issue(self):
        if self.due_date < self.issue_date:
            raise ValueError("due_date must be on or after issue_date")
        return self

```

Validation isn't the only defence, because some dangerous input is perfectly valid. The name `O'Brien` must be accepted, so SQL must always use bound parameters. A note containing `<script>` may be legitimate text, so output must always be escaped for where it's shown. Modern frameworks escape by default; the danger zones are the escape hatches, such as `dangerouslySetInnerHTML`, `innerHTML` and `href` values built from user input. If you must render user-supplied HTML, run it through a maintained sanitiser such as `DOMPurify`, and add a Content Security Policy as a second layer.

When your server fetches a URL

Features that fetch a URL a user supplied, such as importing a logo from a link, sending a webhook to a customer's endpoint, or generating a link preview, turn your server into the user's proxy. Point it at `http://169.254.169.254/`, the address where cloud virtual machines publish their credentials, and it may fetch your cloud keys and hand them back. This is **server-side request forgery**, SSRF, and it was central to one of the largest bank data breaches of 2019.

EXAMPLE `outbound.py` · checking where a URL really points

```

import ipaddress
import socket
from urllib.parse import urlparse

def assert_public_https_url(url: str) -> None:
    parsed = urlparse(url)
    if parsed.scheme != "https" or not parsed.hostname:
        raise UnsafeUrl("only https URLs are allowed")
    for info in socket.getaddrinfo(parsed.hostname, parsed.port or 443):
        address = ipaddress.ip_address(info[4][0])
        if not address.is_global:
            # private, loopback, link-local...
            raise UnsafeUrl(f"{parsed.hostname} resolves to a non-public address")

```

Checking is necessary but not sufficient. An attacker's DNS can answer with a public address when you check and a private one when you connect, a trick called DNS rebinding, so connect to the exact address you checked. Don't follow redirects, or check each one the same way. Set a timeout and a maximum response size. And ideally, make these requests through an isolated outbound proxy with no route to your internal network at all.

Abuse of business flows

Some abuse uses your product exactly as designed. OWASP calls it **unrestricted access to sensitive business flows**. For the invoicing app, think like an abuser:

- **Card testing.** Fraudsters use your payment page to check which stolen cards work, one small payment at a time. Use your provider's fraud tools, rate-limit payment attempts, and add a challenge after failures.
- **Phishing through your domain.** A new account sends convincing "invoices" containing malicious links to thousands of strangers, from your reputable domain. Limit what new accounts can send, check links, and act on complaints quickly.
- **Sign-up and trial farming.** Scripts create accounts to claim free trials or to send spam. Verify email addresses, limit sign-ups, and watch for patterns.
- **Coupon guessing.** Short codes can be brute-forced. Make them long and rate-limit attempts.

Use challenges such as CAPTCHAs sparingly, only on risky actions, and choose ones with accessible alternatives: WCAG 2.2's accessible authentication criterion applies to them too.

COMMON TRAPS

Rate limiting by IP alone. Attackers rotate addresses; offices share them.

Unbounded requests. No limit on body size, page size or batch size.

Nested quantifiers in regular expressions run on long input.

Fetching user URLs from inside your network. SSRF straight to your cloud credentials.

Blocklists. There's always one more dangerous input you didn't list.

Relying on the frontend's validation. The API is the real boundary.

EXERCISES

EXERCISE 16.1 Set the limits

Propose a rate limit, and the key it applies to, for each of these.

- a. Sign-in attempts.
- b. Password-reset requests.
- c. The public "view invoice" page reached from emails.
- d. PDF exports of a year's invoices.
- e. The public API used by integrations.
- f. New sign-ups.

SOLUTION

- a. Per account, about 5 failures in 15 minutes before a growing delay; per IP address, a looser limit such as 50 an hour. Both, because attackers spread attempts across accounts or across addresses.
- b. Per account, about 3 an hour; per IP, about 20 an hour. Always show the same response, whatever the limit.
- c. Per token, generous, such as 60 a minute, since clients may refresh; per IP, a limit that stops enumeration of tokens.
- d. Per team, a few an hour, queued as jobs rather than refused outright, with the quota tied to the plan.
- e. Per API key, using a token bucket such as 100 a minute with bursts, documented and reported in response headers. Higher on higher plans.
- f. Per IP and per email domain, with extra checks when a spike arrives from one source.

The numbers are starting points. Measure real usage, then set limits comfortably above what honest users ever do.

EXERCISE 16.2 Review the logo import

List every problem with this "import logo from URL" feature, and describe the fix.

```
@app.post("/settings/logo-from-url")
def import_logo(body: dict, user=Depends(current_user)):
    response = requests.get(body["url"])
    storage.put(f"logos/{user.team_id}.png", response.content)
    return {"ok": True}
```

SOLUTION

- **SSRF.** Any URL is fetched, including internal addresses and the cloud metadata service. Allow only https, resolve the host, reject non-public addresses, and connect to the address you checked.
- **Redirects are followed,** so a public URL can bounce to an internal one. Disable them, or check every hop.
- **No timeout.** A slow server ties up a worker indefinitely.
- **No size limit.** The response is read whole into memory. Stream it and stop at, say, 5 MB.
- **No content check.** Whatever comes back is saved as a PNG. Check the bytes, then decode and re-encode the image (chapter 13).
- **No authorisation check** that the user may change team settings, and the whole thing runs inside the request rather than a job.

EXERCISE 16.3 Spot the patterns

Explain the danger in each, and fix it.

```
# A: checking that an address belongs to the company domain
COMPANY_EMAIL = re.compile(r"^([a-zA-Z0-9]+)*@example\.com$")

# B: updating a team member
@app.patch("/team/members/{member_id}")
def update_member(member_id: str, body: dict, user=Depends(current_user)):
    db.update_member(member_id, **body)
```

SOLUTION

A has a nested quantifier, `(([a-zA-Z0-9]+)*)`. On an input such as thirty letters followed by `!`, the engine tries an exponential number of ways to split the letters before failing, and one request can hold a CPU core for minutes. Remove the nesting, `^[a-zA-Z0-9]+@example\.com$`, and check the length first.

B has two problems. There's no object-level check, so any user can edit any member of any team. And the body is applied directly, so a member can send `{"role": "owner"}`. Load the member scoped to the user's team, authorise the action, and parse the body into a schema that allows only the fields a member may change.

- ☐ Are there rate limits by user, key, IP and tenant, returning 429 with `Retry-After` ?
- ☐ Is every resource bounded: body size, page size, batch size, time and cost?
- ☐ Are inputs parsed into strict schemas that reject unknown fields?
- ☐ Are user-supplied URLs fetched only after checking the scheme and the resolved address?
- ☐ Have you listed the business flows an abuser would target, and limited them?

GO DEEPER

GUIDE OWASP API Security Top 10, 2023

GUIDE Server-Side Request Forgery Prevention OWASP

GUIDE Input Validation Cheat Sheet OWASP

DRAFT RateLimit header fields for HTTP IETF

Releasing safely

3 exercises

Separate deploying from releasing. Roll out to a few users before all of them, change the database in steps that each work with both the old and the new code, and be able to undo any release in minutes.

Small and often beats big and rare

The Engineer's Craft covered feature flags and the difference between deploying code and releasing behaviour. Build on it with one principle, supported by a decade of DORA's research: small, frequent deployments are safer than large, rare ones. A deploy containing one change is easy to understand, easy to test and easy to reverse. A deploy containing a month of changes is none of those, so it fails more often and takes longer to recover.

Progressive delivery

Release new behaviour to a small group first, watch what happens, and widen gradually: your own team, then 1% of customers, then 5%, 25%, and everyone. Infrastructure can do the same for whole versions, sending a slice of traffic to a **canary** deployment and promoting it only if its error rate and latency match the current version.

Assign users to a rollout deterministically, so that the same person always gets the same answer. Someone whose editor changes between two page loads files a bug report. Hashing the flag name with a stable identifier gives every subject a fixed position between 0 and 100, and raising the percentage only ever adds people.

EXAMPLE rollout.py

```
import hashlib

def in_rollout(flag: str, subject_id: str, percent: float) -> bool:
    """Stable per subject, independent between flags, and monotonic in percent."""
    digest = hashlib.sha256(f"{flag}:{subject_id}".encode()).digest()
    position = int.from_bytes(digest[:8], "big") / 2**64 * 100    # 0 <= position < 100
    return position < percent

# Roll out by team, not by user, so colleagues see the same interface.
if in_rollout("new_invoice_editor", team.id, percent=5):
    return render_new_editor(invoice)
```

Including the flag's name in the hash means the first 5% of one rollout aren't also the first 5% of every other, so a few unlucky customers don't receive every experiment at once. In a business product, roll out per team rather than per person: two colleagues looking at different versions of the same screen is its own source of confusion.

What to watch during a rollout

Decide before starting what would make you stop. Watch technical signals: error rate, latency at the 95th and 99th percentiles, failed jobs. And watch business signals, which catch the bugs that technical ones miss: invoices sent per hour, payments completed, sign-ups finished, support tickets opened. Compare the group that has the change with the group that doesn't. Where you can, make the check automatic, so a bad canary rolls itself back before anyone wakes up.

Database changes: expand and contract

Schema changes are where releases most often go wrong, because code and schema change on different schedules. During a rolling deploy, old and new code run side by side against the same database. And if you roll back the code, you can't roll back a column you've dropped. The safe pattern is **expand and contract**, which Martin Fowler calls parallel change: every step must work with both the code before it and the code after it.

Suppose you're replacing a decimal `total` column with an integer `total_minor`.

RELEASE	SCHEMA	CODE WRITES	CODE READS
1. Expand	Add <code>total_minor</code> , nullable	<code>total</code> and <code>total_minor</code>	<code>total</code>
2. Backfill	Fill <code>total_minor</code> for old rows, in batches	Both	<code>total</code>
3. Switch	Make <code>total_minor</code> required	Both	<code>total_minor</code>
4. Stop writing	No change	<code>total_minor</code>	<code>total_minor</code>
5. Contract	Drop <code>total</code>	<code>total_minor</code>	<code>total_minor</code>

Every row can be deployed on its own, and every row except the last can be rolled back with nothing lost.

EXAMPLE the migrations, in PostgreSQL

```

-- 1. Expand: a new nullable column is a quick metadata change.
ALTER TABLE invoices ADD COLUMN total_minor bigint;

-- 2. Backfill in small batches, repeated until no rows are updated.
UPDATE invoices SET total_minor = round(total * 100)
WHERE id IN (SELECT id FROM invoices WHERE total_minor IS NULL LIMIT 5000);

-- 3. Require it without a long lock: add the check unvalidated, validate it
--     separately, and SET NOT NULL can then use it instead of scanning the table.
ALTER TABLE invoices ADD CONSTRAINT total_minor_present
CHECK (total_minor IS NOT NULL) NOT VALID;
ALTER TABLE invoices VALIDATE CONSTRAINT total_minor_present;
ALTER TABLE invoices ALTER COLUMN total_minor SET NOT NULL;

-- 5. Contract, in a later release, once nothing reads or writes the old column.
ALTER TABLE invoices DROP COLUMN total;

```

Other operations are dangerous on large tables because they take locks that block every read or write while they run. Build indexes with `CREATE INDEX CONCURRENTLY`, which can't run inside a transaction. Add foreign keys and checks as `NOT VALID`, then validate them in a separate step. Avoid changing a column's type in place, which rewrites the whole table; add a new column instead. And start every migration with a `lock_timeout`, so that if a lock isn't available quickly, the migration fails and can be retried, instead of queueing behind a long query while every other request queues behind it.

EXAMPLE

```

SET lock_timeout = '5s';
CREATE INDEX CONCURRENTLY idx_invoices_team_due ON invoices (team_id, due_date);

```

Rollback and roll forward

Keep the previous build and make reverting to it a single command. Keep migrations backward-compatible, as above, so that rolling back the code is always safe. Remember that turning a flag off is the fastest rollback of all. Sometimes rolling forward with a fix is safer than rolling back, for instance once data has been written in a new format, and that's a judgement to make calmly, not at 3 a.m. for the first time. So practise: roll back a harmless release on purpose, and time it.

Tell people what changed

Keep a "what's new" page for users, and give your support team release notes before customers see a change, not after. In a business product, announce significant changes to workflows in advance, and let customers opt in to a beta early. Nobody likes discovering on a Monday morning that the invoice editor moved.

COMMON TRAPS

Big-bang releases straight to every user.

Schema and code changed in one step, so the release can't be rolled back.

Plain `CREATE INDEX` on a big table. Writes stop until it finishes.

Random assignment per request. Users flicker between versions.

Watching only error rates. A bug that stops payments may throw no errors at all.

A rollback nobody has tried. It will fail exactly when you need it.

EXERCISES

EXERCISE 17.1 Plan the migration

Product wants to split `users.name` into `given_name` and `family_name`, with no downtime. Write the sequence of releases. Then consider whether the change is a good idea at all.

SOLUTION

1. Add both columns, nullable.
2. Write all three on every change; keep reading `name`.
3. Backfill in batches, with a best-effort split, and flag names that don't split cleanly so users can correct them.
4. Switch reads to the new columns.
5. Stop writing `name`, and drop it in a later release.

Now the better answer: question the requirement. Many people's names don't divide into a given name and a family name: single names, family names first, several family names, names that change form. Patrick McKenzie's well-known list of falsehoods programmers believe about names starts with exactly this assumption. Keep one full-name field, and add a separate "what should we call you?" field if the product needs a friendly greeting.

EXERCISE 17.2 Plan the rollout

Write a rollout plan for a rebuilt invoice editor: the stages, who is in each, what you watch, and what makes you stop.

SOLUTION

1. **Internal, one week.** Your own team uses it for real invoices.
2. **Beta, one to two weeks.** Customers who opted in, with a feedback link on the page.
3. **5% of teams, then 25%, then 100%,** a few days apart, assigned by team.

Watch the editor's error rate and save latency, the share of invoices created that are sent within a day, support tickets mentioning the editor, and how often beta users switch back to the old editor if you offer that.

Stop and roll back if errors or save latency rise noticeably above the old editor's, if the created-to-sent rate drops by more than a couple of points, or if any bug corrupts or loses invoice data. Data loss means an immediate stop, whatever the numbers say.

EXERCISE 17.3 Which migrations are dangerous?

The `invoices` table has 50 million rows in PostgreSQL. Which of these are dangerous, and how would you make them safe?

- a. `ALTER TABLE invoices ADD COLUMN notes text;`
- b. `ALTER TABLE invoices ADD COLUMN archived boolean NOT NULL DEFAULT false;`
- c. `CREATE INDEX ON invoices (client_id);`
- d. `ALTER TABLE invoices ALTER COLUMN amount TYPE numeric(12,2);`
- e. `ALTER TABLE invoices ADD CONSTRAINT fk_client FOREIGN KEY (client_id) REFERENCES clients (id);`

SOLUTION

- a. **Safe.** A nullable column without a default changes only metadata. Still set a lock timeout, because even a brief lock waits behind long-running queries.
- b. **Safe in PostgreSQL 11 and later,** where a constant default is stored in the catalogue rather than written to every row. In older versions it rewrote the table.
- c. **Dangerous.** A plain index build blocks writes until it finishes. Use `CREATE INDEX CONCURRENTLY`.

- d. **Dangerous.** Changing the type rewrites the whole table under an exclusive lock. Add a new column and migrate with `expand` and `contract`.
- e. **Dangerous as written,** because it checks every row while holding locks. Add it `NOT VALID` , then run `VALIDATE CONSTRAINT` separately, which takes a much weaker lock.

CHECKLIST

- ☐ Can you release a feature without deploying, and switch it off without deploying?
- ☐ Do rollouts start with a small, stable group while you watch agreed metrics?
- ☐ Is every migration compatible with the code that's running when it's applied?
- ☐ Are indexes built concurrently, and lock timeouts set?
- ☐ Have you practised a rollback recently, and timed it?

GO DEEPER

ESSAY [Parallel Change](#) Martin Fowler

DOCS [ALTER TABLE](#) PostgreSQL, including which forms lock and rewrite

ESSAY [Falsehoods Programmers Believe About Names](#) Patrick McKenzie

BOOK [Accelerate](#) Forsgren, Humble & Kim · the research behind small, frequent releases

PART VI

Product and launch

Software that works but that nobody needs is still a failure. The last three chapters are about deciding what to build, finding out whether it worked, and launching and running a product for real users.

Product thinking

3 exercises

Your job isn't to build what's asked for. It's to solve the problem behind the request, with the smallest thing that could work, and to find out quickly whether it did. Engineers who understand the problem make better technical decisions at every level.

Problems before solutions

Requests usually arrive as solutions: "add a button to export invoices as CSV". Behind each one is a problem: "my accountant needs last quarter's invoices every three months". Once you know the problem, better solutions often appear. Here, it might be a read-only link the accountant can use whenever they like, or an integration with their accounting software, either of which removes the chore rather than making it slightly faster.

Clayton Christensen's **jobs to be done** gives this a shape: people "hire" a product to make progress in a particular situation. His best-known example found commuters buying milkshakes in the morning, not because they wanted a milkshake, but because they wanted something to make a long, boring drive more bearable, and a thick shake lasted the whole journey. Freelancers don't want to create invoices. They want to be paid on time without awkward conversations. Every feature of the invoicing app should serve that job.

Talking to users without fooling yourself

Rob Fitzpatrick's short book *The Mom Test* is named for a simple idea: ask questions that even your mother couldn't lie to you about. People are kind, and they'll tell you your idea is great. Compliments, and predictions like "I would definitely use that", are worthless. Evidence comes from what people have actually done.

- **Talk about their life, not your idea.**
- **Ask about specific past events, not about hypothetical futures or general opinions.**
- **Talk less and listen more.** The moment you start pitching, the data stops.

INSTEAD OF

ASK

"Would you use an app that chases late invoices for you?"

"Tell me about the last time a client paid late. What did you do?"

"How much would you pay for this?"

"What do you use today to track who owes you? What does that cost you, in money or time?"

"Do you think automatic reminders are a good idea?"

"When did you last send a reminder? How did you decide when to send it?"

"Would a Slack integration be useful?"

"Where were you, and what were you doing, when you found out the last invoice was paid?"

The strongest signal of all is someone who has already tried to solve the problem: a spreadsheet they built, a tool they pay for, an assistant who does it for them. People who haven't tried to solve a problem usually don't have it badly enough to pay for your solution.

The smallest thing that could work

A minimum viable product isn't a small version of the whole product. It's the smallest experiment that tests your *riskiest assumption*. For automatic payment reminders, the riskiest assumption isn't technical, it's this: do reminders actually get freelancers paid faster? You can test that before writing any scheduling code, by sending reminders by hand for five customers for two weeks and comparing how quickly they're paid. That's a concierge test, and it's often the fastest way to learn.

Basecamp's *Shape Up* adds a useful discipline: decide your **appetite** first, meaning how much time this problem is worth, such as two weeks, and shape a solution that fits it. Fixed time, variable scope, the same principle as the estimation chapter of *The Engineer's Craft*. It stops a modest idea from quietly turning into a quarter's work.

Write it down first

Before building anything substantial, write one page. It forces clarity, and it's far cheaper to argue with a page than with a pull request.

EXAMPLE a one-page spec

Automatic payment reminders

Problem

Freelancers wait a median of 12 days past the due date to be paid. In interviews, 9 of 14 said chasing clients is the part of the job they hate most, and 6 simply don't chase, and wait.

Evidence

Manual reminders sent for 5 customers over 2 weeks: the invoices were paid a median of 4 days sooner than their previous invoices.

Proposal

Polite reminders at 1, 7 and 14 days overdue, on by default, with the wording editable per team and a one-click "pause reminders for this client".

Not doing

Late fees, SMS reminders, escalating tone. Each could come later.

Success looks like

Median days-to-paid for overdue invoices falls from 12 to 7 within 8 weeks of launch. Guardrail: unsubscribes from reminder emails stay under 2%.

Risks

Clients reply to the reminder instead of paying: route replies to the freelancer's inbox.

Amazon's version of this, "working backwards", goes further: write the press release announcing the finished feature, and the answers to customers' likely questions, before any work starts. If the press release isn't exciting, the feature probably isn't worth building.

Saying no

Every feature carries a cost long after it ships: code to maintain, tests to keep green, interface to explain, support questions, and one more thing for every user to understand. Most feature requests should be understood, recorded and declined or deferred, while you watch for the problems that many requests have in common. Say no well: show you understood the problem, explain the trade-off honestly, and offer a workaround where one exists.

Engineers and product

The engineers who have the most influence on a product aren't the ones who implement specifications fastest. They're the ones who understand the users and the business. They read support tickets every week. They sit in on sales and support calls now and then. They notice friction in their own product. They propose simpler alternatives, instrument what they ship, and know how the company makes money. Many companies now hire specifically for this, under the title "product engineer".

COMMON TRAPS

Building exactly what was asked for. The request is a clue, not a specification.

Asking people to predict their own behaviour. "Would you use..." questions produce polite fiction.

MVPs that are just a buggy first version. An experiment has a hypothesis and a result.

Features with no measure of success. You'll never know whether to keep them.

Roadmaps of features rather than problems to solve.

Never removing anything. Unused features still cost you every day.

EXERCISE 18.1 Rewrite the interview questions

Rewrite each question so it asks about past behaviour rather than opinions or predictions.

- a. "Would you pay for automatic late fees?"
- b. "Is invoicing a big pain for you?"
- c. "Would you like a mobile app?"
- d. "What features would you want in an invoicing tool?"

SOLUTION

- a. "Have you ever charged a client a late fee? Walk me through what happened."
- b. "How long did you spend on invoicing last month? What took the most time?"
- c. "When did you last need to do something invoice-related away from your desk? What did you do?"
- d. "Show me how you created and sent your last invoice, step by step." Then watch where they hesitate, work around something, or open another tool.

EXERCISE 18.2 Find the job

For each request, write the question you'd ask the user, a guess at the underlying job, and the smallest way to test it.

- a. "Let me duplicate an invoice."
- b. "Add a dark mode."
- c. "Connect to my bank account."

SOLUTION

- a. **Ask** "When did you last duplicate one, and what did you change?"
Likely job: billing the same client the same amount every month without retyping. **Test:** ship a duplicate button, which is cheap, then measure how often duplicated invoices go to the same client monthly. If most do, the real feature is recurring invoices.

- b. **Ask** "When do you use the app?" **Likely job:** doing admin late at night without the screen glaring, or simple preference. **Test:** respect the operating system's dark setting through CSS; it's cheap enough that a test is barely needed.
- c. **Ask** "What would you do with the bank connection?" **Likely job:** knowing an invoice was paid by bank transfer without checking the bank and marking it by hand. **Test:** a quick "mark as paid" action from the invoice list and the reminder email, then see whether the request still comes up. Bank integrations are expensive and regulated; be sure it's needed first.

EXERCISE 18.3 Write the one-pager

Choose a feature for your own product, or the invoicing app's "client portal" where clients can see all their invoices from one freelancer, and write a one-page spec with the same sections as the example.

SOLUTION

There's no single answer. Check yours against these questions:

- Is the problem stated without mentioning the solution?
- Is the evidence about things people did, not things they said they'd do?
- Does "not doing" contain things someone will genuinely ask for?
- Is success a number with a deadline, plus a guardrail that mustn't get worse?
- Could an engineer who missed every meeting start work from this page?

CHECKLIST

- ☐ Can you state the user's problem without mentioning your solution?
- ☐ Is your evidence about past behaviour rather than opinions or predictions?
- ☐ Is the first version the smallest test of your riskiest assumption?
- ☐ Is there a written success measure, with non-goals?
- ☐ Did you read this week's support tickets?

3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

BOOK **The Mom Test** Rob Fitzpatrick · short, practical, and it will change how you talk to users

BOOK **Shape Up** Ryan Singer, Basecamp · free online

BOOK Inspired Marty Cagan · how strong product teams work

BOOK Competing Against Luck Clayton Christensen et al. · jobs to be done

Measuring what users do

3 exercises

Decide what success looks like before you ship, instrument the few events that measure it, and stay humble about what the numbers can tell you. Most dashboards are noise; a handful of well-defined metrics are gold.

Metrics that mean something

Measure **outcomes**, not output. "Features shipped" and "page views" describe your activity. "Invoices paid on time" describes whether the product is doing its job. Structure the metrics you care about in three layers.

- **One headline metric** that captures the value customers get. For the invoicing app, perhaps the number of invoices paid within their terms each week.
- **A few input metrics** that drive it, which teams can influence directly: sign-ups that send a first invoice, invoices sent per active team, the share of overdue invoices with reminders on.
- **Guardrails:** things that must not get worse while you improve the others, such as error rates, unsubscribes from reminder emails, and support tickets.

Be wary of **vanity metrics**, numbers that only ever go up and make everyone feel good, like total sign-ups since launch. Prefer rates and cohorts, which can go down and so can tell you something.

Activation and retention

Activation is the moment a new user first gets real value. Define it by an action, not a visit: for the invoicing app, "sent a first invoice within seven days of signing up" is much better than "logged in twice". Then look at **retention by cohort**: group users by the week they signed up, and chart what share are still active one, two, four and eight weeks later. If each cohort's curve keeps falling towards zero, people try the product and leave. If the curves flatten out, a core of users has found lasting value, which is one of the clearest early signs of a product people need.

EXAMPLE activation by weekly cohort, in PostgreSQL

```
SELECT date_trunc('week', u.created_at)      AS signup_week,
       count(*)                             AS signups,
       count(*) FILTER (WHERE first_send.at <= u.created_at + interval '7 days')
       AS activated,
       round(100.0 * count(*) FILTER (WHERE first_send.at <= u.created_at + interval '7 days')
            / count(*), 1)                  AS activation_pct
```

```

FROM users AS u
LEFT JOIN LATERAL (
  SELECT min(e.occurred_at) AS at
  FROM events AS e
  WHERE e.user_id = u.id AND e.name = 'Invoice Sent'
) AS first_send ON true
GROUP BY 1
ORDER BY 1;

```

Design events like an API

An analytics event is a contract with everyone who builds a dashboard on it, so design events as carefully as endpoints. Use one naming convention, such as an object followed by a past-tense action: `Invoice Sent`, `Payment Received`, `Client Created`. Give properties consistent names and types. Keep a **tracking plan**, a short document listing each event, when it fires, its properties and its owner, and review changes to it the way you review code.

Send important events from the **server**. Browser analytics are blocked by many extensions and lost when tabs close, so anything you'll make decisions on, such as sign-ups, sends and payments, should be recorded where it happens. Browser events are fine for interface details, such as which tab people open.

EXAMPLE a server-side event

```

analytics.track(
  user_id=user.id,
  event="Invoice Sent",
  properties={
    "invoice_id": invoice.id,
    "team_id": invoice.team_id,
    "currency": invoice.currency,
    "amount_band": amount_band(invoice.total_minor), # a range, not the exact amount
    "line_count": len(invoice.lines),
    "is_first_invoice": invoice.sequence == 1,
  },
)

```

Collect the minimum. No email addresses, names or free text in event properties. Get consent where the law requires it, which in Europe includes most non-essential tracking in the browser. Honour the Global Privacy Control signal, which some US states treat as a legally binding opt-out. Set a retention period. Analytics data is personal data, and it's copied to more tools, with weaker controls, than your database.

Funnels

A funnel follows users through the steps of a flow and shows where they drop out: signed up, added a client, created an invoice, sent it, got paid. The largest drop is usually the most valuable place to work. Then find out *why* people drop there, by watching session recordings with personal data masked, or by asking them, before deciding what to change.

Experiments

An A/B test shows two versions to randomly assigned groups and compares an outcome. Assign by team or user with the stable hashing from chapter 17, using a separate flag for the experiment. Then follow the rules that keep the answer honest.

- **Choose one primary metric and your guardrails before starting.** Picking the metric that happened to move afterwards is how false discoveries are made.
- **Calculate the sample size in advance,** from the current rate and the smallest change worth detecting.
- **Don't peek and stop early.** Checking results daily and stopping the moment they look significant greatly increases the chance of declaring a winner that isn't real. Evan Miller's essay *How Not To Run an A/B Test* explains why.
- **Run whole weeks,** because behaviour on Mondays differs from behaviour on Saturdays.

A rough guide to sample size is Lehr's rule: for 80% power at the usual 5% significance level, you need about $16 \times p(1 - p) \div \delta^2$ subjects in each group, where p is the current rate and δ is the absolute change you want to detect. With a 20% conversion rate and a 2-point change to detect, that's $16 \times 0.2 \times 0.8 \div 0.02^2$, which is 6,400 users per group.

That arithmetic has an important consequence for a young product: you probably don't have the traffic to run useful A/B tests yet. Early on, talk to users, watch them use the product, and make bigger, bolder changes whose effect you can see without statistics.

Numbers and stories

Quantitative data tells you *what* is happening; only people can tell you *why*. Read survey comments, not just scores. Watch a few recorded sessions every week. Keep a running list of recurring support questions. And be suspicious of averages: an overall conversion rate can rise while it falls in every single segment, if the mix of users shifts between segments. That's Simpson's paradox, and segmenting by plan, channel or cohort is the cure.

COMMON TRAPS

Tracking everything. Thousands of events, no decisions.

No tracking plan. `invoiceSent`, `Invoice sent` and `sent_invoice`, all at once.

Personal data in analytics. Emails and names spread across third-party tools.

Peeking at experiments. Stopping when the result looks good produces false winners.

Optimising the metric instead of the goal. Goodhart's law, again.

Averages that hide segments. Simpson's paradox is common, not exotic.

EXERCISES

EXERCISE 19.1 Define the metrics

Define activation, a retention measure, and two guardrails for the invoicing app. Explain each choice in a sentence.

SOLUTION

- **Activation:** a team sends its first invoice to a real client within seven days of signing up. Sending, not just creating, is the moment the product starts doing its job.
- **Retention:** the share of each signup cohort that sends at least one invoice in each later calendar month. Freelancers invoice monthly, so weekly activity would be misleading.
- **Guardrail:** the unsubscribe and complaint rate on reminder emails. Getting freelancers paid mustn't damage their client relationships or your email reputation.
- **Guardrail:** payment failure rate at checkout. A change that speeds up sending but breaks paying is a loss.

EXERCISE 19.2 Clean up the tracking

These events are in production. Rewrite them into a consistent tracking plan, and name anything that shouldn't be tracked at all.

click_button_3	{page: "invoice"}
invoiceSent	{id: 88, clientEmail: "hana@studio.example"}
Invoice sent	{invoice: "inv_8f3k2", total: 1250.00}
payment	{status: "ok", user_email: "ana@example.com"}
New client	{clientName: "Studio North"}

SOLUTION

Invoice Sent	{invoice_id, team_id, currency, amount_band, line_count, is_first_invoice}
Payment Received	{invoice_id, team_id, currency, amount_band, method}
Client Created	{client_id, team_id, source}

- `invoiceSent` and `Invoice sent` are one event under two names, with different id types. Merge them into `Invoice Sent`, and backfill or annotate dashboards.
- `click_button_3` says nothing about intent. Name what the user did, such as `Invoice Preview Opened`, or drop it if nobody will use it.
- Remove `clientEmail`, `user_email` and `clientName`. They're personal data, and ids do the same job.
- Replace exact amounts with bands, and never send money as a float.
- `payment` with a status of "ok" should become separate, clearly named events, such as `Payment Received` and `Payment Failed`.

EXERCISE 19.3 Can you run the test?

You want to test a new onboarding flow. Seven-day activation is currently 30%, and you'd like to detect an improvement to 33%. Using Lehr's rule, how many teams do you need in each group? About 400 teams sign up each week. How long would the test run, and what should you do?

SOLUTION

$16 \times 0.3 \times 0.7 \div 0.03^2 = 3.36 \div 0.0009 \approx 3,733$ teams per group, so about 7,500 in total. At 400 sign-ups a week, that's roughly 19 weeks.

That's too long: the product and the market will change underneath the test. Options: test a bolder change that could plausibly move activation by 6 points or more, which cuts the sample to about a quarter; choose a metric closer to the change with a higher baseline, such as completing the first onboarding step; or skip the experiment, ship the change you believe in, and watch the cohort chart and your user interviews closely.

CHECKLIST

- ☐ Is there one success metric per feature, with guardrails, defined before shipping?
- ☐ Is there a tracking plan with consistent object-action event names?
- ☐ Are important events recorded on the server, without personal data?
- ☐ For experiments: sample size calculated in advance, no peeking, whole weeks?
- ☐ Do you read qualitative feedback alongside the numbers?

GO DEEPER

ESSAY How Not To Run an A/B Test Evan Miller

TOOL Sample size calculator Evan Miller

BOOK Trustworthy Online Controlled Experiments Kohavi, Tang & Xu

BOOK Lean Analytics Alistair Croll & Benjamin Yoskovitz · choosing the metric that matters at each stage

Launch and the first ninety days

3 exercises

Launching is the start of the work, not the end of it. Before real users arrive, make the product observable, recoverable and reachable. Then spend the first ninety days listening, fixing, and resisting the urge to add features.

Configuration and environments

The Twelve-Factor App, a methodology first published by Heroku's engineers in 2011, remains the clearest statement of how to make an application easy to deploy. Its core points for a first launch:

- **Configuration lives in the environment**, not in code, so the same build runs everywhere with different settings.
- **Secrets come from a secret manager** or the platform's encrypted settings, never from the repository, and the development secrets are never the production ones.
- **One build**, promoted from environment to environment, rather than rebuilt for each.
- **Development and production as similar as possible**: the same database engine and version, not SQLite on a laptop and PostgreSQL in production.
- **Logs written to standard output** as a stream of events, collected by the platform.

EXAMPLE config.py · fail at start-up, not at 3 a.m.

```
import os

class Config:
    def __init__(self, env=os.environ):
        self.database_url = env["DATABASE_URL"]           # missing? crash on boot
        self.payments_secret = env["PAYMENTS_SECRET_KEY"]
        self.base_url = env.get("BASE_URL", "http://localhost:8000")
        # Safe defaults: an environment must opt in to real email and live payments.
        self.email_sandbox = env.get("EMAIL_SANDBOX", "true") == "true"
        self.payments_live = env.get("PAYMENTS_LIVE", "false") == "true"
```

Use at least three environments: your laptop; a **staging** environment that mirrors production, with made-up or anonymised data; and **production**. Preview deployments for every pull request are a luxury worth having when your platform makes them easy. And make sure only production can send real email and take real money. The defaults above do that: every other environment has to opt in explicitly.

The minimum operational layer

Stage 12 of the syllabus covers running production properly. Before launch, you need at least this much of it.

NEED	MINIMUM AT LAUNCH	WHY
Error tracking	Every unhandled exception reported, with alerts on new kinds	You hear about bugs before customers email you
Logs	Structured, searchable, with a request id on every line	You can reconstruct what happened to one user
Uptime checks	An outside service checking your health endpoint and one key page	You know you're down before Twitter does
Metrics	Request rate, error rate, latency and queue age	You see things getting worse before they break
Backups	Automatic, and restored at least once, on purpose	An untested backup is a hope, not a backup
Alerts	A handful, each one actionable, routed to a phone	Noisy alerts teach you to ignore all of them
Status updates	A status page, or at least a known place you'll post	Customers forgive outages far more readily than silence

EXAMPLE health checks

```
@app.get("/health")
def health():
    return {"status": "ok", "version": BUILD_VERSION}    # liveness: is the process up?

@app.get("/health/ready")
def ready():
    db.execute("SELECT 1")                               # readiness: can it serve requests?
    return {"status": "ready"}
```

Keep the two separate. If the database is briefly unavailable, the platform should stop sending traffic to your app, based on readiness, but it shouldn't restart it in a loop, based on liveness.

The launch checklist

Go through this in the week before real users arrive. Each item is a sentence, but each has sunk a launch somewhere.

- **Security:** sign-in and recovery reviewed against chapter 5; rate limits on sign-in, reset and sign-up; development secrets rotated out; HTTPS everywhere, with HSTS; security headers, including a Content Security Policy; dependencies scanned for known vulnerabilities.
- **Data:** backups running and one restore tested; every migration reversible; users can export their data and delete their account.
- **Legal and trust:** terms of service and a privacy policy; cookie consent if you use non-essential cookies; a data processing agreement ready for business customers; an accessibility statement if you sell in the EU.
- **Money:** live payment keys configured; webhooks registered in live mode; one real payment made and refunded; tax settings checked.
- **Email:** SPF, DKIM and DMARC in place; every non-production environment unable to email real people; every template read by a human.
- **Support:** a monitored support address; help pages for the five most common tasks; a way to contact you from inside the app.
- **Observability:** error tracking, uptime checks and alerts all tested by deliberately triggering them; a short runbook for your three most likely failures.
- **Performance:** the key pages checked on a mid-range phone; the key endpoints load-tested at ten times your expected traffic.
- **Rollback:** practised, and timed.

The first ninety days

The first week: watch everything. Read every error, every log of a failed payment, every support message. Fix bugs the same day where you can. Talk to every early user you can reach, and watch at least a few of them sign up and send their first invoice, silently, noting every hesitation.

The first month: hold a short weekly review of three things: the activation funnel, new errors, and the themes in support messages. Fix the biggest point of friction each week. Resist new features, however tempting; the product you have isn't finished until new users can succeed with it unaided.

Months two and three: look at retention by cohort. Decide what to double down on and what to remove. Pay down the shortcuts you took in launch week before they turn into incidents. Write a short postmortem for every incident, however small. And notice what maintaining a live product actually involves: dependency updates, security patches, a database that keeps growing, users who find paths you never imagined. That experience is the real deliverable of this stage.

Treat support as part of the product. Fast, human replies build more trust than any feature, and every repeated question is either a missing help page or, more often, a design problem worth fixing so the question never needs asking.

When something breaks

Something will. When it does: say so, on your status page or wherever your users will look. **Mitigate before you diagnose**, by rolling back, switching off a flag or scaling up, because restoring service matters more than understanding the cause in the moment. Then find the root cause, fix it properly, and write a blameless postmortem, as *The Engineer's Craft* describes: what happened, what the impact was, why it happened, what went well, and what will change, with an owner for each action.

COMMON TRAPS

Backups never restored. The first restore attempt shouldn't be during a disaster.

Alerting on everything, or nothing. Both end with nobody responding.

Staging with production credentials. Test invoices emailed to real clients; test charges on real cards.

Secrets in the repository. Rotate anything that was ever committed, even if you deleted it later.

New features in week two while new users still can't finish onboarding.

No way to reach you. Unhappy users who can't complain just leave.

EXERCISES

EXERCISE 20.1 Find the launch blockers

Launch is on Monday. You find the following. Which are blockers, in what order would you fix them, and which can wait?

- Automatic backups are enabled, but nobody has ever restored one.
- Errors are written to logs, but nothing alerts anyone.
- The only alert is "CPU above 80%", sent to an email address nobody reads.
- Payments are still in test mode in production.
- Support email goes to a founder's personal inbox.

- f. Staging uses the production email provider's credentials.
- g. The help centre has two articles.

SOLUTION

1. **(f) first.** Staging can email real customers today. Give it sandbox credentials now.
2. **(d).** In test mode, nobody can actually pay, and everything downstream depends on payment. Switch to live keys, register live webhooks, and make one real payment and refund.
3. **(a).** Restore a backup into a scratch database and check the data. Without that, a bad migration on launch day could be unrecoverable.
4. **(b) and (c) together.** Route error-rate and uptime alerts to a phone, and remove or demote the CPU alert, which says little about what users experience.
5. **(e)** before launch if possible: a shared support address, so messages don't depend on one person's inbox or holiday.
6. **(g) can wait.** Write help articles for the questions people actually ask in the first two weeks.

EXERCISE 20.2 Write three good alerts

Write three alerts for the invoicing app. For each, give the condition, who gets woken, and the first thing they should do.

SOLUTION

1. **The oldest unprocessed payment webhook event is more than 15 minutes old.** Page the on-call engineer. First step: check the worker's health and the dead-letter queue, then the provider's status page. Customers are paying, and invoices aren't being marked paid.
2. **More than 2% of requests to send invoices fail over 10 minutes.** Page on-call. First step: check the last deploy and roll back if it's recent; then check the email provider's status.
3. **The external uptime check fails from two regions for three minutes.** Page on-call. First step: post on the status page, then check the platform and the database.

Everything else, such as a non-empty dead-letter queue or a slowly rising error count, goes to a daily summary rather than a phone. Each alert links to a runbook.

EXERCISE 20.3 Plan your weekly review

Write the agenda for the thirty-minute weekly review you'll hold during your own product's first ninety days, and name the three numbers you'll look at first.

SOLUTION

Weekly product review (30 minutes)

1. Numbers (5 min)
 - Activation: share of last week's sign-ups who sent a first invoice
 - Retention: this month's active teams, by signup cohort
 - Reliability: new error types, incidents, oldest-job alerts
2. Voices (10 min)
 - Top three support themes, with one real quote each
 - One recorded session or user conversation from this week
3. Decisions (10 min)
 - The single biggest point of friction to fix next week
 - Anything to remove, pause or stop measuring
4. Housekeeping (5 min)
 - Postmortem actions: done, or still owned?
 - Launch shortcuts still unpaid

The discipline that matters is the third section: every review ends with one decision about what to fix next, written down, with an owner.

CHECKLIST

- ☐ Is configuration in the environment, secrets in a manager, and does the app fail fast when either is missing?
- ☐ Do you have error tracking, uptime checks, metrics and a few actionable alerts?
- ☐ Have you restored a backup and practised a rollback?
- ☐ Can users reach a person easily, and does someone read what they send?
- ☐ Is there a weekly review of activation, errors and support themes, ending in a decision?

GO DEEPER

GUIDE The Twelve-Factor App

BOOK **Monitoring Distributed Systems** Google's SRE book, including the four golden signals

GUIDE **OWASP Secure Headers Project**

NEXT **Stage 12 of The Engineer's Syllabus** production, infrastructure and reliability in depth

THE TWENTY RULES

One per chapter, each linked back to it. Read them before every launch, and against every pull request that touches one of these areas.

-
- 01 Design from the client's side: nouns for resources, honest status codes, and errors that say how to fix them.
-
- 02 Page with cursors over a total order, and make every write retryable with an idempotency key.
-
- 03 Change APIs additively; when you can't, deprecate out loud, with dates, headers and brownouts.
-
- 04 Pick the API style for your clients, and the simplest real-time channel that works.
-
- 05 Hash passwords with a memory-hard algorithm, keep sessions in locked-down cookies, and make recovery as strong as login.
-
- 06 Use the code flow with PKCE, key users by issuer and subject, and keep tokens out of the browser.
-
- 07 Check every object on every request through one policy, and scope every query to its tenant.
-
- 08 Use the native element for the job, label every input, and keep the main thread free.
-
- 09 Make every flow work by keyboard and screen reader, and reach for ARIA only when HTML can't do it.
-
- 10 Cache server state, put shareable state in the URL, and derive the rest.
-
- 11 Choose rendering per page, measure real users at the 75th percentile, and budget your JavaScript.
-
- 12 Move slow work into idempotent jobs, and write each change and its event in one transaction.
-

- 13 Upload straight to storage, trust bytes rather than names, and serve private files through short-lived signatures.

- 14 Verify, deduplicate, acknowledge fast and reconcile webhooks; authenticate every email you send.

- 15 Never touch a card or trust a price from the browser; fulfil from verified webhooks, idempotently.

- 16 Limit rates, bound every resource, validate at the edge, and check where a URL leads before fetching it.

- 17 Release to a few before all, change schemas in compatible steps, and practise the rollback.

- 18 Understand the problem behind the request, then ship the smallest test of your riskiest assumption.

- 19 Define success before shipping, track a few events well, and never peek at an experiment.

- 20 Launch observable, recoverable and reachable, then spend ninety days listening.

READING LIST

The books behind this volume, and the chapters each one deepens. If you read only one before building, make it *The Mom Test*: it's short, and it will save you from building the wrong thing well.

BOOK	AUTHOR	DEEPENS
Designing Web APIs	Brenda Jin, Saurabh Sahni & Amir Shevat	Ch. 1–4, 14
The Design of Web APIs	Arnaud Lauret	Ch. 1–3
OAuth 2 in Action	Justin Richer & Antonio Sanso	Ch. 6
Inclusive Components	Heydon Pickering	Ch. 9
High Performance Browser Networking	Ilya Grigorik	Ch. 4, 11
Designing Data-Intensive Applications	Martin Kleppmann	Ch. 12, 14
Accelerate	Nicole Forsgren, Jez Humble & Gene Kim	Ch. 17
The Mom Test	Rob Fitzpatrick	Ch. 18
Shape Up	Ryan Singer	Ch. 18
Inspired	Marty Cagan	Ch. 18
Trustworthy Online Controlled Experiments	Ron Kohavi, Diane Tang & Ya Xu	Ch. 19
Lean Analytics	Alistair Croll & Benjamin Yoskovitz	Ch. 19

Compiled September 2026 as the companion to stage 10 of The Engineer's Syllabus, following *The Engineer's Craft*. Research drew on RFC 9110, RFC 9457, RFC 9745, RFC 9700, the IETF's browser-based apps guidance, the Idempotency-Key draft, NIST SP 800-63B-4, OWASP's password storage guidance, the OWASP API Security Top 10, the 2026 WebAIM Million, WCAG 2.2, Core Web Vitals, Standard Webhooks, Google's email sender guidelines, the PCI Security Standards Council, the transactional outbox pattern and Evan Miller on A/B testing.

The self-contained examples, including cursor encoding, idempotency keys, optimistic locking, reset tokens, PKCE against the RFC 7636 test vector, the permission table, webhook verification, file signatures, the token bucket, the SSRF guard, rollout bucketing, currency conversion and the event-loop orderings, were run before publication to confirm they behave as the text says. Examples that depend on a framework, a database server or a payment provider are illustrative. Re-checked in October 2026 with Python 3.14.7, Node 26 and PostgreSQL 18.6: every claim was tested again and held, and chapter 7's account of a missing tenant setting was corrected after testing it on a pooled connection.