API Design Developer Guide, short version

Nov 1, 2026 · @kodebot

This is the short version of the API Design Developer Guide for the team: every rule, what goes wrong without it and what we do instead, with no request examples. The long guide has the same rules with the same numbers, each with the requests that go wrong, the requests that work and the Spring feature used. Read Parts A and B on your first afternoon, after the DDD guide's Parts A and B; C, D and E on day 2; Part F when the situation comes up.

It covers the HTTP endpoints of our Spring Boot microservices and how our Express front end calls them. The examples are the claim microservice and its neighbours, eligibility and payment. Express is the only thing the browser talks to.

Terms

Term	Meaning
Express	Our front-end server. It builds the GOV.UK pages and calls the microservices
Gateway	The server in front of Express. It signs the user in and issues the access token
Version	A number on every record (a claim, a note) that goes up by one on every save; every response with a record includes it
Group of details	Fields the screen shows and saves together, such as contact method, phone and email
Business action	A change the business names as its own step, such as "close claim"
Idempotency key	A random ID sent with a create; a repeat with the same key returns the first result
Access token	The gateway's signed token for the user; its `permissions` claim lists what they may do

Closing a claim, end to end

A caseworker presses Close claim on claim `9f1c`. When Express built the page, it read the claim with the user's token and put the `version` in a hidden field. The browser posts the form to Express, which sends the close action to the claim microservice with that version, the user's token and the trace header. It does not read the claim again: a fresh version would defeat the check. The microservice checks the permission, the version and the rules, and answers `204`; or `409` if the claim changed; or `422` with a code if a rule refuses. Express redirects to the claim page, which reads the claim again. The payment microservice learns of the closure from the `ClaimClosedV1` event, never from Express.

Part A. Endpoints that change data

1. A new record is a create, a named business step is an action, anything else saves a group of details.

An endpoint shaped like a screen lets any screen set the status and skip the closing rules, and changes whenever the screen does. Stop at the first yes: a new record is a `POST` on the collection; a step the business names, or one that needs more than new values, or that sets a field only a rule may set, is a `POST` action; the rest is a `PUT` on a named group of details. Ask the business when unsure.

2. Send one request per save; the microservice saves all of it or nothing.

Two requests for one Check answers page leave the claim without its note when the second times out, and Express cannot undo the first. One request with everything the user collected, saved in one transaction as an approved exception in the design note. Two things the user does at different times are two requests.

3. Save a group of details with `PUT` and send every field in the group; never use `PATCH`.

This is a convention. With partial bodies Express must work out what changed, an empty field means either cleared or not sent, and each microservice decides that for itself. Sending the whole group, with the version, gives one way to save, "empty means cleared" everywhere, rules checked on the whole group, and no comparing in Express: a server-rendered form posts every field anyway. The version check, not the body, prevents a lost update. No group holds a field that only a business action may change.

4. Send an `Idempotency-Key` header with every create, so a repeat cannot create a second record.
A create whose response is lost in a timeout is submitted again, and the claimant has two claims. Express makes a random key when it builds the page, keeps it in a hidden field and sends it with every submit of that page; the repeat returns the first result. A new key on every submit is the same as no key.

5. Send the record's `version` with every change; the microservice refuses an old version with `409`.
Without it, whoever saved last wins and the other change is silently lost. Every change carries the version Express got when it loaded the record, and on `409` Express shows the conflict page. A repeated request also gets `409`, so a change is never made twice.

6. Never retry a change automatically; retry a read once.
A retried change reaches a system without a version check twice. The shared client retries a read once on a timeout or `503` and never retries a change. Against our own microservices a repeat would be refused anyway, so this is also a convention: one behaviour, whatever the target.

7. Return `201` with a `Location` header for a create, and `204` with no body for every other save.
A convention. A body in the save response is a second place to parse the record, which drifts from the `GET`, and after a save Express redirects and the next page reads the record anyway. A create returns `201`, a `Location` header and the new IDs; every other save returns `204`.

8. A change that goes to another system is one business action, and it calls that system once.
Two calls can half happen: the name is changed, the address is not, and nothing undoes the first call. Validate and check permission first, then make one call and change nothing else of our own. Our copy is a cache, and the audit record is written after they confirm.

9. Translate the other system's answer into our own status codes.
Their `412` reaches Express, which has never seen it and shows a generic error. One table in the adapter maps their answers to our codes, and the adapter test proves each row.

Part B. Endpoints that read data

10. Build read endpoints around records, not around screens.
A screen-shaped endpoint is changed, retested and released every time the design moves a panel. The page calls the record endpoints it needs and combines the results.

11. Each kind of record has two response formats only: a summary for lists and a detail for one record.
A convention. A third format for one screen becomes six in six months, each drifting in its own place. The summary is the ID and the few fields a list shows; the detail is every field the caller may see. A list screen's new field goes in the summary, and we accept that the summary grows.

12. Never make one call per row of a list.
Twenty rows become 21 calls, 20 times the time, and the page fails if any one fails. Put the field in the summary, or fetch the rows in one call with a list of IDs, 100 at most.

13. Page, sort and index every list.
An unpaged, unsorted list works on day one and times out a year later, and rows change order between two calls. One page at a time, 100 rows at most, always sorted, and every field a caller may filter on has a database index.

14. Return a lookup list as codes and labels, cached for 5 minutes.
Labels kept in Express go stale when the business renames one, and a retired code shows blank. The microservice returns codes and labels; Express shows the label and stores the code; an inactive value still shows on old records. Five minutes is a choice, made once.

15. Check permission on every endpoint, reads included, even when the caller is our own Express.
Anything on the network can send a header claiming to be anyone and read every claimant's details. Each endpoint declares the permission it needs, takes the user from the access token, and a test proves a user without it gets `403`. The build fails on an endpoint with no permission. Hiding a button is not security.

Part C. Processes across several microservices

16. A process across microservices has one owning microservice, and Express calls only that one.
When Express runs the steps, a failure after the first leaves the claim closed and the payments running, and nobody owns the half-done job. Express makes one call to the owner, which records the progress and raises an alert when a process is stuck. Each such process needs design group approval and an architecture decision record.

17. Change data in another microservice by sending an event, never by calling its API.
An HTTP call that times out leaves us not knowing whether the change happened. An event, stored with our change, is delivered until the other microservice processes it. Before its first change, a microservice may still call another to read or check something.

18. Save each step's change and its event in one transaction, and skip an event you already processed.
An event sent outside the transaction may never be sent, and a broker may deliver it twice. The outbox on the sending side and the inbox of processed event IDs on the receiving side cover both, because a crash can happen between any two lines.

19. Decide what happens when a step fails before you build the process.
When only the happy path is designed, the failure event arrives, the listener logs it, and the claimant is paid for a closed claim. The failure path is a named business step agreed with the business, because a done step cannot be rolled back.

20. Link the steps by events each microservice reacts to; a directing microservice only for many steps or deadlines.
A two-step process does not need a state machine, and a five-step chain of events cannot be followed. Start with each microservice reacting to an event and sending its own. Switch to one directing owner, sending command messages and waiting for reply events, past the line we chose once: more than three steps, a deadline, or progress the business must see.

21. Return `204` and a progress field when the user needs only the first step; `202` with a status URL when they must wait for all steps.
A plain `204` says "Awarded" while the letter is unsent and hides a failed step. Close claim needs only the first step, so `204` with a progress field; award claim must be complete, so `202` with a status URL that Express checks on refresh. Which answer a process gives is decided once, in its design note.

Part D. Conventions

22. `GET` reads, `POST` creates or acts, `PUT` saves a group; never `PATCH`, and no `DELETE` on business records.
A `GET` that writes is triggered by a browser prefetch, and a `DELETE` erases the audit trail. One meaning per method: a record ends with an action such as close or cancel; `DELETE` only when the design group agrees a record may be removed for ever; no `PATCH` because a group is saved whole (rule 3).

23. Name collections with plural nouns and actions with lower-case verbs; nest one level at most.
A path that mirrors the object tree needs three IDs to fetch one attachment, and someone parses a year out of an ID. IDs are opaque UUIDs that Express passes back unchanged; filters, sorting and paging are query parameters. The naming is our choice, made once.

24. Return a JSON object with camelCase fields, ISO dates, UTC times and two-decimal money.
A bare array cannot grow a `totalItems` field, a day-month date is ambiguous, and floating-point money prints two different totals. An object, ISO dates, UTC times ending in `Z`, money with two decimals; camelCase is a choice, made once.

25. Never put personal data in a URL; search by personal details with `POST /claims/search`.
The gateway, Express, the microservice and the load balancer all log URLs, so a National Insurance number in a query string lands in four places our data protection rules do not cover. Personal details go in the body of a `POST` that only reads.

26. Return the same status codes and the same error body from every microservice.
Express has one error handler, and it only works if every microservice answers the same way; a feature that writes its own `400` shows a blank error summary. A domain error goes through one shared handler to an RFC 9457 problem details body: a status, a `code` Express turns into text, a developer `detail` never shown, a `traceId`, and never personal data.

27. Make a breaking API change in three releases, and update `openapi.yaml` in the same pull request as the code.
Express and the microservices release on different days, so a one-release rename shows blank names for two days. Add the new field, move the callers, then remove the old field; or add `/v2` beside `/v1` with `Deprecation` and `Sunset` headers. The build compares the contract with the released version and fails on a breaking change, and Express generates its client from it.

Part E. The Express front end

28. The browser never calls a microservice; Express calls it with the user's token.
Page JavaScript calling a microservice puts the token in the browser, where any script can read it, and makes the address public. The browser holds only the gateway's session cookie; Express accepts requests only from the gateway and forwards the token and the `traceparent` header.

29. Read from several microservices in parallel, but send each save to one microservice.
Three reads one after another make a 1.2-second page, and a form that writes to two microservices has nobody owning the half-saved result. Start all reads at once, each allowed to fail on its own, and send each save as one request to one microservice.

30. Give every call from Express a time limit and a circuit breaker.
One slow microservice with no time limit uses up Express's connections and hangs the whole site. All calls go through one shared client with time limits; microservices keep a request under four seconds so they give up first; after several failures the breaker fails fast for a short time.

31. Show the page with the data that loaded; show an error page only when essential data failed.
Failing the page because a panel the caseworker did not need failed stops their work. Decide which data is essential for each page and show the rest with a message; one error handler turns status and code into what the user sees.

32. Keep answers in the Express session until the save button, then save once and redirect.
Saving a draft on every page fills the microservice with half-finished claims that passed no rules, and rendering the result directly lets a refresh submit the form again. Answers stay in the server-side session, the save button sends one request, then Express redirects.

33. Cache lookup lists only; never cache business records.
A cached claim shows as open after another caseworker closed it, and the user gets a `409` they could not avoid. Keep a lookup list for as long as its `Cache-Control` header says, and read business records fresh every time.

Part F. Situations you will meet

34. Upload a file as multipart, store the bytes in an object store, keep only a reference on the record, and stream downloads through Express.
A file inside the record makes every read carry 8 MB, and a public download link in a browser history opens a payslip without a sign-in. Multipart with a size limit and the record's version; the bytes go to the object store before the transaction opens; a reference stays on the record; downloads go through Express with the user's token and permission.

35. Return field errors as `400` with an `errors` list of field, code and message; keep business-rule refusals as `422`.
One message for everything leaves the user hunting for the mistake. One entry per field, with a stable code Express turns into GOV.UK text. A rule refusal is `422` and shown differently, because the user cannot fix it on the page. Two status codes for that is a choice, made once.

36. A bulk action is one request per record; add a batch endpoint only when the business names the batch as one action, with a result per item.

One all-or-nothing request lets one claim with an open note stop all 40, and nobody can tell which 39 were fine. Express sends one close per claim with its version and shows a result per row. When the business names the batch, it becomes a record with its own action answering `202` and a status URL, still one claim per transaction. Starting from Express is our choice: it needs no new endpoint.

37. Store a "save and return later" draft as a record of its own, with no business rules until the final create.

The Express session is for one sitting, so a claimant who returns tomorrow finds every answer gone. A draft record in the microservice, saved like a group of details, may be incomplete and wrong until the real create, which is where the rules live.

38. On `409`, show the user the fresh record and a way to start again; never retry with the new version.

Express quietly retrying with the new version overwrites the other caseworker's change, and nobody knows. A conflict page says what happened and shows the current values; only the user can decide whose change wins.

39. A microservice with no user present calls another with its own client-credentials token and permission, never a shared key or a stored user token.

A shared key lives in three repositories and a chat message, cannot be rotated without a coordinated release, and opens every endpoint. The microservice signs in as itself and gets a short-lived token with only the permission it needs, which the receiving endpoint checks like any other. A user's token belongs to that user's request.

40. An outside callback gets its own endpoint, which checks the signature, stores it with its ID, answers `202` and raises our own event; no business logic there.

A handler that does the work lets anyone who finds the URL verify evidence, and a slow save makes the provider time out and resend. Check the signature, store the callback with its ID and its outbox entry in one transaction, answer `202`, and let a listener on our own event translate their IDs and call the business action.

41. Export CSV from `GET .../export` with the list's filters, a row limit and the personal-data permission; a large export returns `202` and a status URL.

An export built from list pages takes minutes, pulls every field including National Insurance numbers, and times out at the gateway. One endpoint beside the list, same filters, streamed, with a row limit; personal-data columns only with the export permission; an export the user cannot wait for answers `202` with a status URL.

42. Show another microservice's data by reading it in Express; store a copy only when a rule needs it.

A copy of the claimant's name kept for display is wrong for years after they change it. Express reads the claim, then the claimant from its owner, and composes the page. The claim microservice stores only the eligibility decision its rules need and the claimant's ID.

43. Remove a record created by mistake with a `cancel` action and a reason; keep it, hide it from lists by default, still return it on `GET`.

A deleted claim leaves the audit trail pointing at nothing, and an event other microservices received with no matching close. Cancel records the mistake with a reason; lists hide cancelled records by default; `GET` still returns them.

44. When a system the save depends on is down, answer `503` with `Retry-After`; Express says "try again later" and does not retry the change.

A convention. A `500` counts as a bug in alerts and tells the user nothing; `503` says "nothing was saved, try later", and alerts can tell an outage next door from a bug of ours. Express never retries a change; for a read it shows the page with what loaded. The circuit breaker (rule 30) protects the other system, whichever code we return.

45. Make `openapi.yaml` the contract: the microservice validates every response against it, and Express's client tests run against its examples.

A contract nobody runs says `openedAt` while the microservice still sends `createdAt`, and nobody finds out until every row shows "Invalid Date". A microservice test checks every response against the file, and Express's client tests run against the file's examples. A rename then fails a build, not a page.