API Design Developer Guide

Oct 31, 2026 · @kodebot

This guide says how we design the HTTP endpoints of our microservices, and how our Express front end calls them. The microservice examples are Spring Boot 4 (Spring MVC and Spring Security 7), with the Spring feature named under each one. Each rule starts with a situation you will meet, shows the code that goes wrong, says what happens, and shows the code that works. The examples use the claim microservice, where a **caseworker** creates a **claim** for a **claimant**, adds **evidence** and **notes**, and closes the claim. The **eligibility** microservice says whether a person may claim. The **payment** microservice pays claimants. Express is the only thing the browser talks to.

Your first two days. Read the walkthrough and Parts A and B (rules 1 to 15) on your first afternoon, after the Pragmatic DDD Guide's Parts A and B; Parts C, D and E on day 2. Part F is not for reading through: it is the answers for the situations you will meet, one at a time, when you meet them.

The rules

One line per rule. The section with the same number has the situation, the requests and the reasons.

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.
2. Send one request per save; the microservice saves all of it or nothing.
3. Save a group of details with `PUT` and send every field in the group; never use `PATCH`.
4. Send an `Idempotency-Key` header with every create, so a repeat cannot create a second record.
5. Send the record's `version` with every change; the microservice refuses an old version with `409`.
6. Never retry a change automatically; retry a read once.
7. Return `201` with a `Location` header for a create, and `204` with no body for every other save.
8. A change that goes to another system is one business action, and it calls that system once.
9. Translate the other system's answer into our own status codes.

Part B. Endpoints that read data

10. Build read endpoints around records, not around screens.
11. Each kind of record has two response formats only: a summary for lists and a detail for one record.
12. Never make one call per row of a list.
13. Page, sort and index every list.
14. Return a lookup list as codes and labels, cached for 5 minutes.
15. Check permission on every endpoint, reads included, even when the caller is our own Express.

Part C. Processes across several microservices

16. A process across microservices has one owning microservice, and Express calls only that one.
17. Change data in another microservice by sending an event, never by calling its API.
18. Save each step's change and its event in one transaction, and skip an event you already processed.
19. Decide what happens when a step fails before you build the process.
20. Link the steps by events each microservice reacts to; a directing microservice only for many steps or deadlines.

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.

Part D. Conventions

22. `GET` reads, `POST` creates or acts, `PUT` saves a group; never `PATCH`, and no `DELETE` on business records.
23. Name collections with plural nouns and actions with lower-case verbs; nest one level at most.
24. Return a JSON object with camelCase fields, ISO dates, UTC times and two-decimal money.
25. Never put personal data in a URL; search by personal details with `POST /claims/search`.
26. Return the same status codes and the same error body from every microservice.
27. Make a breaking API change in three releases, and update `openapi.yaml` in the same pull request as the code.

Part E. The Express front end

28. The browser never calls a microservice; Express calls it with the user's token.
29. Read from several microservices in parallel, but send each save to one microservice.
30. Give every call from Express a time limit and a circuit breaker.
31. Show the page with the data that loaded; show an error page only when essential data failed.
32. Keep answers in the Express session until the save button, then save once and redirect.
33. Cache lookup lists only; never cache business records.

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.
35. Return field errors as `400` with an `errors` list of field, code and message; keep business-rule refusals as `422`.
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.
37. Store a "save and return later" draft as a record of its own, with no business rules until the final create.
38. On `409`, show the user the fresh record and a way to start again; never retry with the new version.
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.
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.
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.
42. Show another microservice's data by reading it in Express; store a copy only when a rule needs it.
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`.
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.
45. Make `openapi.yaml` the contract: the microservice validates every response against it, and Express's client tests run against its examples.

Terms

Term	Meaning
Express	Our front-end server. It builds the GOV.UK pages on the server and calls the microservices
Gateway	The server in front of Express. It signs the user in and keeps the session
Record	One thing a microservice stores, such as a claim or a note
Version	A number on every record 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 instead of a second record
Access token	The gateway's signed token (a JWT) for the user; its `permissions` claim lists what they may do
`traceparent`	The header that carries the trace ID through every call, so one action can be followed through every log
Owner	The one microservice that starts a process across microservices and records its progress
Design group	The lead engineer, senior engineers and business experts, who approve design decisions

Closing a claim, end to end

A caseworker opens claim `9f1c` and presses **Close claim**. This is what happens, in order. Later rules point back at these six requests.

1. When Express built the claim page, it read the claim with the user's token and put the `version` in a hidden field of the form.

```
GET /v1/claims/9f1c
Authorization: Bearer eyJ...
traceparent: 00-4bf92f35...

200 OK
{ "claimId": "9f1c", "version": 7, "status": "OPEN", "claimType": "STANDARD",
  "createdAt": "2026-10-01T09:15:00Z", "paymentsStopped": false }
```

2. The browser posts the form to Express. The browser only ever talks to Express, and only carries the gateway's session cookie.

```
POST /claims/9f1c/close HTTP/1.1
Host: express.example.gov.uk
Cookie: session=...

version=7
```

3. Express sends the business action to the claim microservice, with the version the user saw. It does not read the claim again: a fresh version would defeat the check.

```
POST /v1/claims/9f1c/close
Authorization: Bearer eyJ...
traceparent: 00-4bf92f35...
Content-Type: application/json

{ "version": 7 }
```

4. The claim microservice checks permission, loads the claim, checks the version, applies the rules, saves the claim and a `ClaimClosedV1` event in one transaction, and answers with no body.

```
204 No Content
```

If a rule refuses, the answer is an error body with a `code` Express knows:

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

{ "status": 422, "title": "Business rule violated", "code": "CLAIM_HAS_OPEN_ACTIONS",
  "detail": "Claim 9f1c has 2 notes that need action", "traceId": "4bf92f35..." }
```

5. Express redirects the browser to a confirmation page. A refresh of that page cannot close the claim twice.

```
303 See Other
Location: /claims/9f1c/closed
```

6. Later, the payment microservice receives `ClaimClosedV1` from the message broker, stops the payments and sends `PaymentsStoppedV1`. The claim microservice receives that and sets `paymentsStopped: true`. Express never calls the payment microservice for this.

Every call from Express to a microservice carries the `Authorization: Bearer` and `traceparent` headers of step 1, and every microservice path starts with `/v1`. The rest of this guide leaves all three out, so each example shows only its point.

Part A. Endpoints that change data

1. Pick the endpoint by what the user saves: a new record is a create, a named business step is an action, anything else saves a group of details

You are building the **Change contact details** pages, which end with a Check answers page, and the **Close claim** button.

Bad — one endpoint shaped like the screens:

```
PUT /claims/9f1c
{ "version": 7, "status": "CLOSED", "contactMethod": "EMAIL", "emailAddress": "jane@example.com",
  "phoneNumber": "", "claimType": "STANDARD", "fullName": "Jane Doe", ... }
```

Any screen can now set `status` to `CLOSED` and skip the closing rules, and every change to a screen changes the request body.

Good — the endpoint follows what the user saves. Stop at the first yes:

1. Does the save create a new record? **Create**: `POST /claims`.
2. Is it a business action? Yes if the business names it as a step (close, verify evidence, change address), if it needs more than new values (an effective date, a reason), or if it sets a field only a rule may set (`status`, `closedAt`). **Business action**: `POST /claims/9f1c/close`.
3. Otherwise it changes a group of details. **Save details**: `PUT /claims/9f1c/contact-preferences`.

```
POST /claims/9f1c/close
{ "version": 7 }

PUT /claims/9f1c/contact-preferences
{ "version": 7, "contactMethod": "EMAIL", "phoneNumber": "", "emailAddress": "jane@example.com" }
```

If you are not sure about step 2, ask the business. A complex record has 10 to 20 such endpoints.

Why: the business decides the endpoints, so the API stays stable while screens change. Also bad: `POST /claims/9f1c/update-phone` for one question page; the save is the group, not the page.

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

You are building **Create claim**: five question pages, then a Check answers page that creates the claim with its first note.

Bad — Express sends one request per thing created:

```
const { claimId } = await claims.post('/claims', { claimantId, claimType });
await claims.post(`/claims/${claimId}/notes`, { text, category, needsAction: true });
```

The second call times out. The claim exists without its note, Express cannot undo the first call, and the user sees an error for work that was half saved.

Good — one request with everything the user collected; the microservice saves it in one database transaction (an approved exception in its design note: Pragmatic DDD Guide, rule 27):

```
POST /claims
Idempotency-Key: 7d3e...
{ "claimantId": "c88", "claimType": "STANDARD",
  "notes": [ { "text": "Call back Monday", "category": "CONTACT", "needsAction": true } ] }

201 Created
Location: /claims/9f1c
{ "claimId": "9f1c", "noteIds": ["4ab2"] }
```

`POST /claims` accepts at most 10 notes, so the transaction stays small. 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`

You are saving the **contact preferences** group: contact method, phone number, email address.

Bad — Express sends only the field the user typed:

```
PATCH /claims/9f1c
{ "emailAddress": "jane@example.com" }

200 OK
```

Express must work out which fields changed, so it keeps the old values in the form or the session and compares. An empty phone field now means either "cleared" or "not sent", and each microservice decides that for itself. And because the request patches the whole claim, a later screen patches `status` and closes a claim without the rules (rule 1).

Good — every field in the group, changed or not, with the record's `version`:

```
PUT /claims/9f1c/contact-preferences
{ "version": 7, "contactMethod": "EMAIL", "phoneNumber": "", "emailAddress": "jane@example.com" }

204 No Content
```

The microservice compares each field with the stored value, checks the rules, saves all changes together, audits the changed field names (not the values) and publishes one event listing the changed fields. Each field belongs to one group, and a group never holds a field that only a business action may change.

Why: this is a convention. The `version` check (rule 5) stops a lost update either way. Sending the whole group 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. Once chosen, no microservice accepts a partial body.

4. Send an `Idempotency-Key` header with every create, so a repeat cannot create a second record

You are building the Check answers page that calls `POST /claims`.

Bad — a create with no key:

```
POST /claims
{ "claimantId": "c88", "claimType": "STANDARD" }
```

The claim is saved, but the response is lost in a timeout. Express shows an error, the user presses the button again, and claimant `c88` has two claims.

Good — Express makes a random ID when it builds the page, keeps it in a hidden field, and sends it every time that page is submitted:

```
<input type="hidden" name="idempotencyKey" value="7d3e...">
```

```
POST /claims
Idempotency-Key: 7d3e...
{ "claimantId": "c88", "claimType": "STANDARD" }

201 Created
Location: /claims/9f1c
{ "claimId": "9f1c" }
```

The repeat with the same key returns the same `201` and the same `claimId`. Adding an item stored inside a record (`POST /claims/9f1c/evidence`) is a change to that record, so it sends the `version` instead.

Why: a request can time out after the record was saved, and only the key lets the microservice recognise the repeat. A new key generated 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`

Two caseworkers open claim `9f1c` at version 7. One closes it. The other, a minute later, saves new contact details.

Bad — the second save carries no version:

```
PUT /claims/9f1c/contact-preferences
{ "contactMethod": "EMAIL", "phoneNumber": "", "emailAddress": "jane@example.com" }

204 No Content
```

The microservice cannot know this user saw an old screen. Whatever the second user saw wins, and the first user's change is silently lost.

Good — every request that changes an existing record carries the `version` Express got when it loaded the record:

```
PUT /claims/9f1c/contact-preferences
{ "version": 7, "contactMethod": "EMAIL", "phoneNumber": "", "emailAddress": "jane@example.com" }

409 Conflict
Content-Type: application/problem+json
{ "status": 409, "title": "Record changed", "code": "CONCURRENT_UPDATE",
  "detail": "Claim 9f1c is at version 8; the request was made from version 7", "traceId":
"4bf92f35..." }
```

Express shows the 409 page (rule 38), with the latest version, and the user makes the change again. A repeated request also gets `409`, so a change is never made twice.

Why: the version is the only proof that the user saw the current record. "Last save wins" is a lost change waiting to happen.

6. Never retry a change automatically; retry a read once

You are writing the shared HTTP client Express uses for every microservice call.

Bad — one retry rule for everything:

```
async function call(req) {
  try { return await send(req); }
  catch (e) { return await send(req); }   // retry everything once
}
```

`POST /customers/42/change-address` times out after the other system saved it. The retry sends it again. For a system without a version check, the address is changed twice, or the second attempt fails on a record that already moved on.

Good — the method decides:

```
async function call(req) {
  try { return await send(req); }
  catch (e) {
    const isRead = req.method === 'GET' || req.path.endsWith('/search');
    if (isRead && (e.timeout || e.status === 503)) return await send(req);
    throw e;                                      // a change is never retried
  }
}
```

If a change fails or times out, Express shows the error and the user checks the record before submitting again. A read (`GET`, or `POST /claims/search`) is retried once on a timeout or `503`.

Why: a repeated change may be made twice, and a repeated read is harmless. Against our own microservices a repeat would be refused by the version check or answered by the idempotency key, so this is also a convention: one behaviour in the shared client, whatever the target.

7. Return `201` with a `Location` header for a create, and `204` with no body for every other save

You are writing the response of `PUT /claims/9f1c/contact-preferences`.

Bad — the whole claim comes back after a save:

```
PUT /claims/9f1c/contact-preferences
{ "version": 7, ... }

200 OK
{ "claimId": "9f1c", "version": 8, "status": "OPEN", "contactMethod": "EMAIL", ...all fields... }
```

Every save now has its own response shape, and Express has one more place to parse a claim, which drifts from `GET /claims/9f1c`.

Good — a create returns `201 Created`, a `Location` header with the new record's URL, and the new IDs; every other save returns `204 No Content` with an empty body:

```
POST /claims
201 Created
Location: /claims/9f1c
{ "claimId": "9f1c", "noteIds": ["4ab2"] }
```

```
PUT /claims/9f1c/contact-preferences
204 No Content
```

If Express needs the record afterwards, it reads it with `GET /claims/9f1c`.

Why: a convention. After a save Express redirects, and the next page reads the record again (rule 32), so a body in the save response would be parsed in a second place and read by nobody. Once chosen, no save returns a body.

8. When a change goes to another system, make one business action per change and call that system once

The claimant's name and address live in the central customer system, not in our database. You are building **Change name and address**.

Bad — one endpoint sends both changes:

```
public void updateDetails(CustomerId id, UpdateDetails req) {
    customers.changeName(id, req.fullName());       // call 1: succeeds
    customers.changeAddress(id, req.address());     // call 2: times out
    audit.record("details updated", id);
}
```

The name is changed in the other system, the address is not, the endpoint returns an error, and nothing can undo call 1.

Good — each change is its own business action. The microservice validates and checks permission first, then makes exactly one call and changes nothing else:

```
GET /customers/42/address
200 OK
{ "customerId": "42", "version": "v7", "line1": "2 Low Road", "town": "York", "postcode": "YO1
7HH" }

POST /customers/42/change-address
{ "version": "v7", "line1": "1 High Street", "town": "Leeds", "postcode": "LS1 1AA" }
204 No Content
```

Here `version` is the other system's ETag, text not a number, passed on in an `If-Match` header. A copy we keep is a cache, updated after the other system confirms. The audit record is written after the confirmation. If the other system has no version, the last save wins; write that risk in the ADR.

Why: a single call either happened or did not; two calls can half happen.

9. Translate the other system's answer into our own status codes

You are writing the adapter that calls the customer system in rule 8.

Bad — the other system's answer goes straight through:

```
catch (HttpClientErrorException e) {
    throw new ResponseStatusException(e.getStatusCode(), e.getResponseBodyAsString());   // 412
reaches Express
}
```

Express gets `412 Precondition Failed`, a code it has never seen, and shows a generic error page instead of the 409 page.

Good — one table, applied in the adapter:

The other system	We return
Refuses the change	`422` with our own code, such as `ADDRESS_REJECTED`
Says the version changed (`412`)	`409` with code `CONCURRENT_UPDATE`
Is unreachable or answers `503`	`503`. Nothing was changed
Does not answer in time	`504` with code `OUTCOME_UNKNOWN`. The change may or may not be made. Express tells the user to check before trying again

```
@Component
class CustomerClient {
    private final RestClient http;        // built once, with the base URL, the bearer token and
the time limits

    void changeAddress(CustomerId id, Address address, String etag) {
        try {
            http.post().uri("/customers/{id}/change-address", id.value())
                .header("If-Match", etag).body(address)
                .retrieve().toBodilessEntity();
        } catch (HttpClientErrorException e) {
            if (e.getStatusCode().value() == 412) throw new ConcurrentUpdate(id);     // -> 409
CONCURRENT_UPDATE
            if (e.getStatusCode().value() == 422) throw new AddressRejected(id);       // -> 422
ADDRESS_REJECTED
            throw e;
        } catch (HttpServerErrorException e) {
            throw new DependencyUnavailable("customers");                              // -> 503;
nothing was changed
        } catch (ResourceAccessException e) {                                          // no
connection, or no answer in time
            if (e.getCause() instanceof ConnectException) throw new
DependencyUnavailable("customers");   // never reached them -> 503
            throw new OutcomeUnknown(id);                                               //
reached them, reply lost -> 504 OUTCOME_UNKNOWN
        }
    }
}
```

Spring: `RestClient` throws `HttpClientErrorException`, `HttpServerErrorException` and `ResourceAccessException`; the adapter turns each into our own error. A `@WireMockTest` proves each row

of the table: WireMock plays the customer system and answers `412`, `422`, `503` and a delay.

Why: Express handles one set of codes for every microservice.

Part B. Endpoints that read data

10. Build read endpoints around records, not around screens

You are building the **claim summary page**: the claim, its latest three notes, and the payment status.

Bad — one endpoint for the page:

```
GET /claims/9f1c/summary-page
200 OK
{ "heading": "Claim 9f1c", "notesPanel": [...], "paymentBadge": "Paid", "showCloseButton": true }
```

The design team moves the notes to their own tab. The microservice changes, is retested and is released for a screen change.

Good — the page calls the record endpoints it needs and combines the results. Every read endpoint is one of five kinds:

Kind	Example	Rule
One record	`GET /claims/9f1c`	Detail format; `404` if missing
A list with filters	`GET /claims?status=OPEN&page=0&size=20`	Rule 13
A record's children	`GET /claims/9f1c/notes?page=0&size=3&sort=createdAt,desc`	`404` if the claim is missing, not an empty list
A lookup list	`GET /reference-data/CLAIM_TYPE`	Rule 14
A calculated answer	`GET /claims/9f1c/entitlement?from=2026-01-01&to=2026-01-31`	The one kind built for a question; the microservice applies the business rules

```
const [claim, notes] = await Promise.all([
  claims.get(`/claims/${id}`),
  claims.get(`/claims/${id}/notes?page=0&size=3&sort=createdAt,desc`),
]);
```

Why: a screen-shaped endpoint changes every time the screen changes; a record-shaped one does not.

11. Give each kind of record two response formats only: a summary for lists and a detail for one record

The **dashboard** needs the claim type and the caseworker's name in its list of claims.

Bad — a third format for that one screen:

```
record ClaimForDashboard(String claimId, String claimType, String caseworkerName, boolean
```

```
overdue) {}
```

Six months later there are `ClaimForDashboard`, `ClaimForSearch` and `ClaimForReport`, each parsed in its own place in Express, each drifting.

Good — two formats, and the list screen's new field goes in the summary:

```
record ClaimSummary(String claimId, long version, String claimType, String status,
                    Instant createdAt, String caseworkerName) {}          // for lists
record ClaimDetail(String claimId, long version, String claimType, String status,
                   Instant createdAt, String caseworkerName, ContactPreferences
contactPreferences,
                   List<EvidenceItem> evidence, boolean paymentsStopped) {} // for one record
```

Summary is the ID and the few fields a list shows. **Detail** is every field the caller may see.

Why: a convention. Two formats have a price, the summary grows with every list screen's needs, and we accept it. A third format for one screen ties the microservice to that screen.

12. Never make one call per row of a list

The list of open claims shows 20 rows, and each row needs the number of notes that need action.

Bad — one call per row:

```
const list = await claims.get('/claims?status=OPEN&sort=createdAt,desc&page=0&size=20');
for (const row of list.items) {
  const notes = await claims.get(`/claims/${row.claimId}/notes?needsAction=true&page=0&size=1`);
// 20 more calls
  row.openActions = notes.totalItems;
}
```

The page makes 21 calls, takes 20 times as long, and fails whenever any one of them fails.

Good — add the field to the summary format, or fetch the rows in one call:

```
GET /claims?status=OPEN&sort=createdAt,desc&page=0&size=20
200 OK
{ "items": [ { "claimId": "9f1c", "version": 7, "status": "OPEN", "openActions": 2, ... }, ... ],
  "page": 0, "size": 20, "totalItems": 48213 }

GET /claims?ids=9f1c,7a2d,...
```

`ids` takes 100 IDs at most.

Why: one call fails once and takes one round trip.

13. Page, sort and index every list

You are building `GET /claims` with a filter on status.

Bad — the whole table, in whatever order the database returns it:

```
GET /claims
200 OK
{ "items": [ ...48,000 claims... ] }
```

It works on the first day. A year later it times out, and the rows change order between two calls, so the user misses some.

Good — one page at a time, always sorted, every filter field indexed:

```
GET /claims?status=OPEN&sort=createdAt,desc&page=0&size=20
200 OK
{ "items": [ ... ], "page": 0, "size": 20, "totalItems": 48213 }
```

`size` is 100 at most. Every field a caller may filter on has a database index.

Why: an unindexed, unpaged list gets slower as the data grows until it times out.

14. Return a lookup list as codes and labels, cached for 5 minutes

The **claim type** dropdown needs the list of claim types.

Bad — the labels live in Express:

```
const CLAIM_TYPES = { STANDARD: 'Standard claim', URGENT: 'Urgent claim' };   // in Express
```

The business renames "Urgent claim" to "Priority claim". The microservice is updated, Express still shows the old label, and old records with a retired type show a blank.

Good — the microservice returns codes and labels, and Express stores the code:

```
GET /reference-data/CLAIM_TYPE
200 OK
Cache-Control: max-age=300
{ "items": [ { "code": "STANDARD", "label": "Standard claim", "active": true },
             { "code": "URGENT",   "label": "Priority claim", "active": true },
             { "code": "LEGACY",   "label": "Legacy claim",   "active": false } ] }
```

Express shows the `label` and stores the `code`. A value with `active: false` still shows on old records but is not offered in forms.

Why: labels change and codes never do. Five minutes is a choice, made once: long enough to take the load off the microservice, short enough that a renamed label shows within minutes.

15. Check permission on every endpoint, reads included, even when the caller is our own Express

You are adding `GET /claims/9f1c`, a read, called only by Express.

Bad — the endpoint trusts a header and skips the check because it is "only a read":

```
@GetMapping("/claims/{id}")
public ClaimDetail get(@PathVariable String id, @RequestHeader("X-User-Id") String userId) {
    return service.get(id);          // no permission check; the header is whatever the caller
sent
}
```

Anything on the network can send `X-User-Id: admin` and read every claimant's details.

Good — the endpoint declares the permission it needs, takes the user from the access token, and a test proves a user without the permission gets `403`:

```
@RestController
@RequestMapping("/claims")
class ClaimController {
    @GetMapping("/{id}")
    @PreAuthorize("hasAuthority('VIEW_CLAIM')")         // the authorities come from the access
token
    ClaimDetail get(@PathVariable ClaimId id) {          // the string becomes our type at the
edge (DDD guide, rule 6)
        return claimQueries.detail(id);
    }
}

@WebMvcTest(ClaimController.class)
class ClaimControllerTest {
    @Autowired MockMvc mvc;

    @Test
    @WithMockUser(authorities = "EDIT_CLAIM")          // signed in, but without this permission
    void refusesUserWithoutPermission() throws Exception {
        mvc.perform(get("/claims/9f1c")).andExpect(status().isForbidden());
    }
}
```

Spring: `@PreAuthorize` with `@EnableMethodSecurity` checks the authority before the method runs; `@WebMvcTest` and `@WithMockUser` prove it. The authorities exist only because a `JwtAuthenticationConverter` bean maps the token's `permissions` claim (Pragmatic DDD Guide, "One-time setup"); by default Spring maps only `scope`, as `SCOPE_...`.

The build checks that no endpoint is missing a `@PreAuthorize`. The microservice refuses even when the call comes from our own Express app. Hiding a button the user cannot use helps the user but is not security.

Why: reads return personal data, and anything on the network can call the endpoint.

Part C. Processes across several microservices

A database transaction cannot cover two microservices. Step 6 of the walkthrough is the example for this part: closing a claim must also stop its payments, which live in the payment microservice.

16. When a process spans several microservices, one microservice owns it and Express calls only that one

You are building **Close claim**, which must also stop payments.

Bad — Express runs the process:

```
await claims.post(`/claims/${id}/close`, { version });        // succeeds
await payments.post(`/payments/stop`, { claimId: id });        // times out
```

The claim is closed, the payments keep going, Express shows an error, and nobody owns the half-done job.

Good — Express makes one call to the owner, and the owner records the progress:

```
POST /claims/9f1c/close
{ "version": 7 }
204 No Content

GET /claims/9f1c
200 OK
{ "claimId": "9f1c", "status": "CLOSED", "paymentsStopped": false, ... }
```

Express shows "Waiting for payments to stop" while `paymentsStopped` is false. An alert is raised when a process is stuck. Every such process needs design group approval and an ADR.

17. Change data in another microservice by sending an event, never by calling its API

Inside the close-claim process, the claim microservice must make the payment microservice stop paying.

Bad — an HTTP call inside the process:

```
claim.close(openActions, today);
claims.save(claim);
paymentsClient.post("/payments/stop", claim.id());    // times out: stopped or not?
```

The call times out. The claim is closed, and nobody knows whether the payments stopped.

Good — the change is an event, stored with the claim and delivered until the payment microservice processes it:

```
claim.close(openActions, today);        // raises ClaimClosed inside the entity
claims.save(claim);                     // saves the claim and ClaimClosedV1 together (rule 18)
```

```
{ "eventId": "c1d2...", "type": "ClaimClosedV1", "claimId": "9f1c", "closedOn": "2026-10-31" }
```

Before its first change, a microservice may still call another to read or check something, such as asking eligibility before creating a claim.

Why: an HTTP call that times out leaves you not knowing whether the change happened; an event is delivered until it is processed.

18. Save each step's change and its event in one transaction, and skip an event you already processed

You are writing the listener in the payment microservice that reacts to `ClaimClosedV1`.

Bad — the event is sent outside the transaction, and the listener trusts it arrives once:

```
claims.save(claim);                     // transaction commits
broker.publish(new ClaimClosedV1(id));  // process crashes here: closed claim, no event

// the broker calls this for each ClaimClosedV1
void on(ClaimClosedV1 e) { payments.stopFor(e.claimId()); }   // runs twice on redelivery
```

Either the event is never sent and the payments never stop, or the broker delivers it twice and the second run creates a second "stopped" record.

Good — an outbox on the sending side, an inbox on the receiving side:

```
// sender, in the claim microservice: claim and event in one transaction; a background sender
publishes afterwards
@Transactional
public void close(ClaimId id, long expectedVersion) {
    Claim claim = claims.findById(id).orElseThrow(() -> new ClaimNotFound(id));
    claim.requireVersion(expectedVersion);
    claim.close(noteService.countNeedingAction(id), LocalDate.now(clock));
    claims.save(claim);          // @Version: an old version throws
OptimisticLockingFailureException;
                                 // @DomainEvents: the Outbox listener stores ClaimClosedV1 in
the same transaction (DDD guide, rule 19)
    audit.record("claim closed", id);
}

// receiver, in the payment microservice: called by the message listener for each ClaimClosedV1
@Transactional
public void on(ClaimClosedV1 e) {
    if (inbox.seen(e.eventId())) return;                                    // seen it: skip
(the Inbox class: DDD guide, rule 21)
    inbox.record(e.eventId());                                              // same
transaction; its unique _id catches a race
    paymentService.stopFor(new ClaimId(e.claimId()), e.closedOn());        // their string,
our type
}
```

Spring: `@Transactional` with a `MongoTransactionManager` bean wraps both writes (DocumentDB 4.0 or later); `@DomainEvents` on the aggregate and an `@EventListener` write the outbox entry; the `Inbox` class wraps `MongoTemplate`.

Why: an event can be delivered more than once, and a crash can happen between any two lines.

19. Decide what happens when a step fails before you build the process

You are designing the close-claim process, and the payment microservice may be unable to stop a payment that is already with the bank.

Bad — the design only names the happy path:

```
ClaimClosedV1 -> payments stop -> PaymentsStoppedV1 -> paymentsStopped = true
```

`PaymentStopFailedV1` arrives. Nobody wrote what to do, so the listener logs it and the claimant is paid for a closed claim.

Good — the failure path is a named business step, agreed with the business and the design group:

```
ClaimClosedV1 -> payments stop -> PaymentsStoppedV1    -> paymentsStopped = true
                               \-> PaymentStopFailedV1  -> create a caseworker task "Recover
payment"
```

Why: step 2 cannot be undone once step 3 fails, so a new business action, not a rollback, handles the failure.

20. Link the steps by events that each microservice reacts to; use one directing microservice only for many steps or deadlines

You have two processes: "close claim, then stop payments" (two steps) and "award a claim" (create the award, set up payments, send a letter, each within a deadline).

Bad — a directing owner for the two-step process:

```
claim microservice: send command StopPayments -> wait for PaymentsStopped -> set paymentsStopped
                    (plus a timer, a retry policy and a state table, for one step)
```

The simple process gets a state machine nobody needed; or the reverse, the five-step award process is a chain of events nobody can follow.

Good — start with each microservice listening for an event, doing its step and sending its own event (choreography):

```
ClaimClosedV1 -> payment microservice stops -> PaymentsStoppedV1 -> claim microservice updates
```

Switch to one directing owner (orchestration) when there are more than three steps, deadlines, or the business must see each step's progress. The owner then sends each step as a command message, not an HTTP call, and waits for the reply event:

```
award owner: CreateAward -> AwardCreated; SetUpPayments -> PaymentsSetUp; SendLetter ->
LetterSent
             status: IN_PROGRESS (step 2 of 3), deadline 5 working days
```

Why: a short chain of events is simple; a long one becomes impossible to follow. Where the line sits is a judgement we made once: more than three steps, a deadline, or progress the business must see. Do not move it per process.

21. Return `204` and a progress field when the user needs only the first step; return `202` with a status URL when the user must wait for all steps

Close claim: the caseworker only needs the claim closed; payments stop later. Award claim: the caseworker must know when the award is complete.

Bad — both answer `204` and say nothing more:

```
POST /claims/9f1c/award
204 No Content
```

The screen says "Awarded" while the letter has not been sent, and the caseworker has no way to see that step 3 failed.

Good — `204` with a progress field when the first step is enough; `202` with a status URL when the user must wait:

```
POST /claims/9f1c/close
204 No Content
GET /claims/9f1c  ->  { ..., "paymentsStopped": false }

POST /claims/9f1c/award
202 Accepted
Location: /claims/9f1c/award

GET /claims/9f1c/award
200 OK
{ "status": "IN_PROGRESS", "step": "SET_UP_PAYMENTS" }       // later COMPLETED, or FAILED with a
reason
```

Express shows the status and checks again on refresh.

Why: the user should never wait for work they do not need, and never be told "done" when it is not. 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 caseworker wants to remove a claim that was opened by mistake.

Bad — the record is deleted, and a read changes data on the way:

```
DELETE /claims/9f1c
204 No Content

GET /claims/9f1c/mark-as-read       // a GET that writes
```

The audit trail for the claim is gone, and a link prefetch by the browser marks claims as read.

Good — one meaning per method:

```
GET  /claims/9f1c                     reads; never changes anything
POST /claims                          creates
POST /claims/9f1c/cancel              acts; the record and its history stay (rule 43)
PUT  /claims/9f1c/contact-preferences saves a group
POST /claims/search                   reads by personal details (rule 25)
```

To end a record, use an action such as `close` or `cancel`. `DELETE` only when the design group agrees a record may be removed for ever. "Never `PATCH`" is rule 3's convention, a group is saved whole, not a failure of its own.

23. Name collections with plural nouns and actions with lower-case verbs; nest one level at most

You are adding attachments to notes.

Bad — the path mirrors the whole object tree, and the ID carries meaning:

```
GET /claims/9f1c/notes/4ab2/attachments/CLM-2026-000731-A
```

Express has to know three IDs to fetch one attachment, and someone parses `2026` out of the ID.

Good — plural nouns, lower-case verbs, one level of nesting, opaque UUIDs:

```
GET  /claims/9f1c/notes/4ab2
POST /claims/9f1c/resolve-action
GET  /attachments/0c7e...
GET  /claims?status=OPEN&sort=createdAt,desc&page=0&size=20
```

Filters, sorting and paging are query parameters. IDs are opaque strings, UUIDs without dashes (shortened in these examples), that Express passes back unchanged and never reads meaning from.

Why: a predictable URL needs no lookup. Opaque IDs prevent a real failure; plural nouns, lower-case verbs and one level of nesting are our naming choice, made once.

24. Return a JSON object with camelCase fields, ISO dates, UTC times and two-decimal money

You are writing the response of `GET /claims/9f1c/payments`.

Bad — a bare array, mixed field styles, local times and floating-point money:

```
[ { "payment_id": "p1", "PaidOn": "31/10/2026 09:15", "amount": 242.8599999 } ]
```

Express's generated client cannot add a `totalItems` field to an array later, the date is ambiguous between UK and US order, and the amount prints as £242.86 on one page and £242.85 on another.

Good — an object, camelCase, ISO dates, UTC times ending with `Z`, money with two decimals:

```
{ "items": [ { "paymentId": "p1", "paymentDate": "2026-10-31", "paidAt": "2026-10-31T09:15:00Z",
               "amount": 242.86 } ] }
```

We use plain JSON, not JSON:API; the GDS standards suggest JSON:API only for an organisation with no standard of its own.

Why: Express's generated client needs one shape. Objects, ISO dates and decimal money prevent real errors; camelCase is a choice, made once.

25. Never put personal data in a URL; search by personal details with `POST /claims/search`

The caseworker searches for a claimant by National Insurance number.

Bad — the number is in the query string:

```
GET /claims?nino=QQ123456C
```

The gateway, Express, the microservice and the load balancer all write the URL to their access logs, and the number is now in four places the data protection rules do not cover.

Good — personal details go in the body of a `POST` that only reads:

```
POST /claims/search
{ "nino": "QQ123456C" }

200 OK
{ "items": [ { "claimId": "9f1c", ... } ] }
```

Names and dates of birth go the same way, never in a path or query string.

Why: URLs are written to logs.

26. Return the same status codes and the same error body from every microservice

You are handling the case where the claim has notes that need action when the user closes it.

Bad — the feature writes its own error response:

```
catch (ClaimHasOpenActions e) {
    return ResponseEntity.status(400).body(Map.of("error", "Cannot close: " + e.getMessage()));
}
```

Express gets `400` and looks for field errors to show next to fields, finds none, and shows a blank error summary; the notes microservice answers the same situation with `409` and a different body.

Good — the feature throws a domain error; one shared error handler maps each kind of error to its status and body:

```
if (notesNeedingAction > 0) throw new ClaimHasOpenActions(id, notesNeedingAction);   // in
Claim.close

@RestControllerAdvice
class ApiErrors extends ResponseEntityExceptionHandler {         // the one error handler of the
microservice; rules 44 and 9 add 503 and 504
    @ExceptionHandler(RuleRefused.class)                          // every business rule error
extends RuleRefused
    ProblemDetail ruleRefused(RuleRefused e) { return problem(UNPROCESSABLE_CONTENT, "Business
rule violated", e.code(), e.getMessage()); }

    @ExceptionHandler({VersionConflict.class, OptimisticLockingFailureException.class})   // the
entity's check, and @Version's
    ProblemDetail conflict(Exception e) { return problem(CONFLICT, "Record changed",
"CONCURRENT_UPDATE", e.getMessage()); }

    @ExceptionHandler(NotFound.class)
    ProblemDetail notFound(NotFound e) { return problem(NOT_FOUND, "Not found", e.code(),
e.getMessage()); }

    @ExceptionHandler(AccessDeniedException.class)              // @PreAuthorize said no: 403
here, so the catch-all cannot make it a 500
    ProblemDetail forbidden(AccessDeniedException e) { return problem(FORBIDDEN, "Forbidden",
"FORBIDDEN", "You do not have permission for this action"); }

    @ExceptionHandler(Exception.class)                            // anything else is a bug
    ProblemDetail bug(Exception e) {
        log.error("unhandled", e);                                // the stack trace goes to the
log, never to the caller
        return problem(INTERNAL_SERVER_ERROR, "Unexpected error", "INTERNAL_ERROR", "Quote the
traceId to the team");
    }

    @Override                                                     // Spring MVC's own errors (bad
JSON, unknown URL, wrong method) get the same two fields
    protected ResponseEntity<Object> handleExceptionInternal(Exception e, Object body,
HttpHeaders headers, HttpStatusCode status, WebRequest request) {
        ResponseEntity<Object> r = super.handleExceptionInternal(e, body, headers, status,
request);
        if (r != null && r.getBody() instanceof ProblemDetail p) { p.setProperty("code",
HttpStatus.valueOf(status.value()).name()); p.setProperty("traceId", Trace.current()); }
        return r;
    }

    private ProblemDetail problem(HttpStatus status, String title, String code, String detail) {
        ProblemDetail p = ProblemDetail.forStatusAndDetail(status, detail);
        p.setTitle(title);
        p.setProperty("code", code);
        p.setProperty("traceId", Trace.current());
        return p;
    }
}
```

```
422 Unprocessable Content
Content-Type: application/problem+json
{ "status": 422, "title": "Business rule violated", "code": "CLAIM_HAS_OPEN_ACTIONS",
  "detail": "Claim 9f1c has 2 notes that need action", "traceId": "4bf92f35..." }
```

Spring: a `@RestControllerAdvice` returning `ProblemDetail` is written as `application/problem+json`. Extending `ResponseEntityExceptionHandler` brings Spring MVC's own errors through the same class: `handleExceptionInternal` is their last stop, and rule 35 overrides the `400` for field errors. The status names are static imports of `HttpStatus`.

Status	Meaning	What Express does
`400`	Invalid request; `errors` names each wrong field	Shows each error next to its field
`401`	Not signed in or session ended	Sends the user to sign in through the gateway
`403`	Signed in but not allowed	Shows "You do not have permission"
`404`	Record does not exist	Shows a not-found page
`409` `CONCURRENT_UPDATE`	Record changed after the screen loaded	Shows the 409 page (rule 38); never retries
`409` `REQUEST_IN_PROGRESS`	The first request with this `Idempotency-Key` is still running	Asks the user to wait, then check the record
`422`	A business rule refused the change; `code` names the rule	Shows the message for that `code`
`422` `IDEMPOTENCY_KEY_REUSED`	Same key, different request: an Express bug	Shows an error page
`500`	Unexpected error	Shows an error page with the `traceId`
`503` `DEPENDENCY_UNAVAILABLE`	A needed system is unavailable; nothing changed; `Retry-After` says when to try (rule 44)	Shows "Try again later"; retries a read once, never a change
`504` `OUTCOME_UNKNOWN`	Another system did not answer in time (rule 9)	"We could not confirm your change. Check before you try again." Never retries

Every error body is RFC 9457 problem details. Express picks the message from `code`; `detail` is for developers and never shown; `traceId` finds the request in every log. Error bodies never contain personal data.

Why: Express has one error handler for every microservice, and it only works if every microservice answers the same way.

27. Make a breaking API change in three releases, and update `openapi.yaml` in the same pull request as the code

The business wants `fullName` on the eligibility microservice's `GET /claimants/c88` split into `firstName` and `lastName`.

Bad — one release:

```
- "fullName": "Jane Doe"
+ "firstName": "Jane", "lastName": "Doe"
```

The microservice is released on Tuesday, Express on Thursday. For two days every claim page shows no claimant name.

Good — three releases, and the contract file in the same pull request as the code:

```
Release 1 (microservice): returns fullName AND firstName, lastName; accepts either on create
Release 2 (Express):      reads and sends firstName, lastName
Release 3 (microservice): removes fullName
```

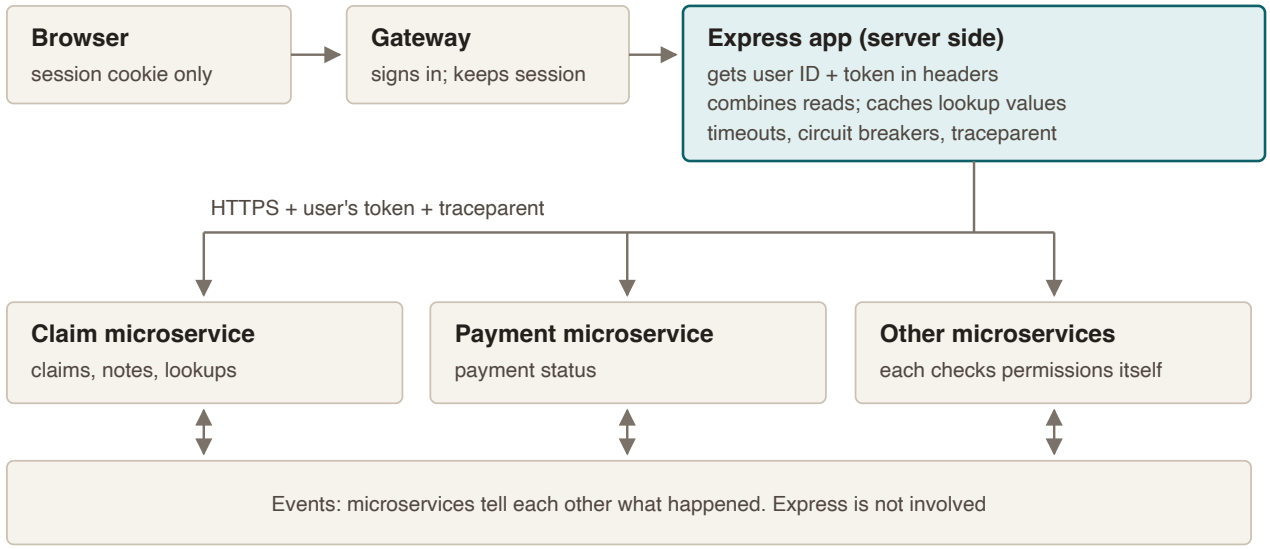
```
# contracts/openapi.yaml, same pull request as release 1
fullName:  { type: string, deprecated: true }
firstName: { type: string }
lastName:  { type: string }
```

Safe at any time: a new endpoint, an optional request field, a new response field (Express ignores unknown fields). Breaking: removing, renaming or retyping a field, making it required, changing a URL. Every path starts with `/v1`, as in the walkthrough. If three releases are impossible, add `/v2` beside `/v1` with `Deprecation` and `Sunset` headers on `/v1`. The build compares `contracts/openapi.yaml` with the released version and fails on a breaking change, and Express generates its client from the file, so a removed field does not compile.

Why: Express and the microservices are released at different times.

Part E. The Express front end

Every browser request goes to the gateway, which signs the user in and passes the request to Express with the user's ID and access token in headers. Express builds the page on the server and calls the microservices. So Express is already our backend-for-frontend (BFF), a server-side layer built for one front end, and we add no other layer. If a mobile app comes later, the design group decides whether it needs its own BFF.



Browser → Express → microservices: the browser talks only to Express, and Express calls each microservice. The gateway signs the user in and passes the user's ID and token to Express. Express calls each microservice over HTTPS with the user's token and traceparent, and each microservice checks permissions itself. The microservices tell each other what happened with events, and Express is not involved.

28. The browser never calls a microservice; Express calls it with the user's token

A developer wants the notes panel to refresh without reloading the page.

Bad — page JavaScript calls the microservice:

```
// in the browser
const notes = await fetch('https://claims.internal/v1/claims/9f1c/notes',
                          { headers: { Authorization: 'Bearer ' + token } });
```

The access token is now in the browser, where any script on the page can read it, and the microservice address is public.

Good — the browser calls Express, and Express calls the microservice with the token from the gateway:

```
// in Express
app.get('/claims/:id/notes', async (req, res) => {
  const notes = await claims.get(`/claims/${req.params.id}/notes?page=0&size=20`, {
    headers: { Authorization: `Bearer ${req.gateway.accessToken}`, traceparent: req.traceparent
},
  });
  res.render('notes', { notes });
});
```

The browser holds only the gateway's session cookie. Express accepts requests only from the gateway, which strips any identity headers the browser sends. Every call carries the standard `traceparent` header, so one action can be followed through every log.

Why: only Express knows the microservice addresses, and the token never reaches a place we do not control.

29. Read from several microservices in parallel, but send each save to one microservice

The claim page shows the claim, its notes and the payment status, from two microservices.

Bad — reads one after another, and a form that saves to two microservices:

```
const claim = await claims.get(`/claims/${id}`);                 // 400 ms
const notes = await claims.get(`/claims/${id}/notes`);           // + 300 ms
const pay   = await payments.get(`/payments?claimId=${id}`);     // + 500 ms, throws if down

await claims.put(`/claims/${id}/contact-preferences`, body);
await payments.post(`/payments/${id}/change-bank-details`, body); // half saved if this fails
```

The page takes 1.2 seconds, three calls one after another, and the form writes to two microservices with nobody owning the half-saved result (rule 16).

Good — all reads at once, each allowed to fail on its own; each save one request to one microservice:

```
const [claim, notes, pay] = await Promise.allSettled([
  claims.get(`/claims/${id}`),
  claims.get(`/claims/${id}/notes`),
  payments.get(`/payments?claimId=${id}`),
]);
```

Why: parallel reads keep the page fast; a single save keeps the data consistent (rule 16).

30. Give every call from Express a time limit and a circuit breaker

The payment microservice is slow this morning.

Bad — a call with no time limit:

```
const pay = await fetch(`${PAYMENTS}/payments?claimId=${id}`);   // waits for ever
```

Every claim page waits for the payment call, Express runs out of connections, and the whole site hangs because one microservice is slow.

Good — all calls go through one shared HTTP client module with time limits and a circuit breaker:

```
const client = httpClient({
  timeouts: { read: 2000, change: 5000 },   // ms
  circuitBreaker: { failures: 5, openFor: 30000 },
});
```

Microservices keep a request, with their own retries, under 4 seconds, so they give up first. After several failures the breaker stops calling that microservice for a short time and fails at once.

31. Show the page with the data that loaded; show an error page only when essential data failed

The payment microservice is down, and a caseworker opens a claim.

Bad — any failed call fails the page:

```
const [claim, notes, pay] = await Promise.all([...]);   // one rejection throws
res.render('claim', { claim, notes, pay });              // never reached
```

The caseworker sees a blank error page and cannot work on the claim, because a panel they did not need failed.

Good — decide which data is essential for the page, and show the rest with a message:

```
const [claim, notes, pay] = await Promise.allSettled([...]);
if (claim.status === 'rejected') return errorPage(res, claim.reason);   // essential
res.render('claim', {
  claim: claim.value,
  notes: notes.status === 'fulfilled' ? notes.value : null,
  payments: pay.status === 'fulfilled' ? pay.value : { unavailable: true },  // "Payment details
are not available right now"
});
```

One error handler turns status and `code` into what the user sees (rule 26).

32. Keep answers in the Express session until the save button, then save once and redirect

You are building the five question pages of **Create claim**.

Bad — each page saves a draft to the microservice, and the final page renders the result directly:

```
app.post('/create-claim/claim-type', async (req, res) => {
  await claims.put(`/claims/${req.session.draftId}/claim-type`, req.body);   // a draft claim per
page
  res.render('next-question');
});
```

The microservice fills with half-finished claims; the user presses F5 on the last page and the form posts again.

Good — answers stay in the server-side session; the save button sends one request; then redirect:

```
app.post('/create-claim/claim-type', (req, res) => {
  req.session.claim = { ...req.session.claim, claimType: req.body.claimType };
  res.redirect(303, '/create-claim/first-note');
});

app.post('/create-claim/check-answers', async (req, res) => {
  const { claimId } = await claims.post('/claims', req.session.claim,
                                        { headers: { 'Idempotency-Key': req.body.idempotencyKey }
});
  delete req.session.claim;
  res.redirect(303, `/claims/${claimId}/created`);  // confirmation page shows the reference
});
```

Pages work without JavaScript: every form posts to Express, and the Change links on the Check answers page edit the session. Field errors: rule 35. A `409`: rule 38. Save and come back later: rule 37.

Why: a redirect after the save means a refresh cannot submit the form twice.

33. Cache lookup lists only; never cache business records

Express calls `GET /reference-data/CLAIM_TYPE` on every page, and someone wants to speed up `GET /claims/9f1c` the same way.

Bad — a cache in front of a business record:

```
const claim = await cache.get(`/claims/${id}`, { ttl: 60 });   // 60-second cache
```

Another caseworker closes the claim; this user sees it as open for a minute, submits a change with version 7, and gets a `409` they could not have avoided.

Good — keep a lookup list for as long as its `Cache-Control` header says, and read business records fresh every time:

```
const types = await cache.get('/reference-data/CLAIM_TYPE');     // honours Cache-Control: max-age=300
const claim = await claims.get(`/claims/${id}`);                 // never cached
```

Why: users must always see the latest version of a record.

Part F. Situations you will meet

34. Send an evidence file as multipart to the claim microservice, store the bytes in an object store, keep only a reference on the record, and stream downloads through Express

A caseworker uploads a payslip to claim `9f1c`.

Bad — the file goes into the record as text, and the download is a public link:

```
POST /claims/9f1c/evidence
Content-Type: application/json
{ "fileName": "payslip.pdf", "content": "JVBERi0xLjQKJ..." }      // 8 MB of base64

200 OK
{ "downloadUrl": "https://files.example.gov.uk/9f1c/payslip.pdf" }  // anyone with the link
```

The claim record is now 8 MB, every `GET /claims/9f1c` carries it, and a link in someone's browser history opens a claimant's payslip without a sign-in.

Good — multipart with a limit and the record's `version`, bytes to the object store, a reference on the record, and the download goes through Express with the user's token:

```
POST /claims/9f1c/evidence
Content-Type: multipart/form-data; boundary=----x
------x
Content-Disposition: form-data; name="version"
7
```

```
------x
Content-Disposition: form-data; name="file"; filename="payslip.pdf"
Content-Type: application/pdf
(bytes, at most 10 MB)

201 Created
Location: /claims/9f1c/evidence/e3a1
{ "evidenceId": "e3a1", "fileName": "payslip.pdf", "size": 812331 }
```

```
@PostMapping(path = "/{id}/evidence", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
@PreAuthorize("hasAuthority('EDIT_CLAIM')")
ResponseEntity<EvidenceCreated> upload(@PathVariable ClaimId id,
                                       @RequestParam long version,
                                       @RequestParam MultipartFile file) throws IOException {
    FileName fileName = new FileName(file.getOriginalFilename());
    EvidenceId evidenceId = claimService.addEvidence(id, version, fileName,
file.getInputStream());
    return ResponseEntity.created(URI.create("/claims/" + id.value() + "/evidence/" +
evidenceId.value()))
                         .body(new EvidenceCreated(evidenceId.value(), fileName.value(),
file.getSize()));
}

// application.properties
// spring.servlet.multipart.max-file-size=10MB
// spring.servlet.multipart.max-request-size=11MB
```

`claimService.addEvidence` sends the bytes to the object store through one adapter, before it opens the transaction (DDD guide, rule 17), and keeps only a `FileReference` on the evidence item (DDD guide, rule 10); a DocumentDB document holds 16 MiB at most, so the bytes never go into the record. The adapter is tested against Floci, which plays the object store locally.

Spring: `MultipartFile` receives the part, and `spring.servlet.multipart.max-file-size` refuses a bigger one before the controller runs.

```
// Express: GET /claims/:id/evidence/:evidenceId/file
app.get('/claims/:id/evidence/:evidenceId/file', async (req, res) => {
  const upstream = await
claims.stream(`/claims/${req.params.id}/evidence/${req.params.evidenceId}/file`);
  res.set('Content-Disposition', `attachment; filename="${upstream.fileName}"`);
  upstream.pipe(res);
});
```

The claim microservice checks `VIEW_CLAIM` on the file endpoint like any other (rule 15), and the file name in the response comes from the record, never from the URL.

Why: a record stays small and a file is only ever read by a signed-in user with the permission.

35. Return field errors as `400` with an `errors` list of field, code and message; keep business-rule refusals as `422`

On the contact preferences page the caseworker picks EMAIL, leaves the email address blank and types letters in the phone number.

Bad — one message for everything:

```
400 Bad Request
{ "error": "Validation failed: emailAddress, phoneNumber" }
```

Express cannot tell which field has which problem, so it shows one vague error at the top and the user hunts for the mistake.

Good — one entry per field, with a stable code Express can turn into GOV.UK text. The constraint's message is the code; the text comes from one message table:

```
record ContactPreferencesRequest(
    @NotNull(message = "REQUIRED") Long version,
    @NotNull(message = "REQUIRED") ContactMethod contactMethod,
    @Pattern(regexp = "[0-9 ]*", message = "FORMAT") String phoneNumber,
    @NotBlank(message = "REQUIRED") String emailAddress) {}

@PutMapping("/{id}/contact-preferences")
@PreAuthorize("hasAuthority('EDIT_CLAIM')")
ResponseEntity<Void> saveContactPreferences(@PathVariable ClaimId id, @Valid @RequestBody
ContactPreferencesRequest body) {
    claimService.saveContactPreferences(id, body.version(), body.toContactPreferences());
    return ResponseEntity.noContent().build();
}

// in ApiErrors (rule 26): the override that shapes every 400
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException e,
HttpHeaders headers,
                                                              HttpStatusCode status, WebRequest
request) {
    ProblemDetail p = ProblemDetail.forStatusAndDetail(status, e.getFieldErrorCount() + " fields
are invalid");
    p.setTitle("Request is invalid");
    p.setProperty("code", "INVALID_REQUEST");
    p.setProperty("traceId", Trace.current());
    p.setProperty("errors", e.getFieldErrors().stream()
        .map(f -> Map.of("field", f.getField(), "code", f.getDefaultMessage(), "message",
messages.text(f)))
        .toList());
    return ResponseEntity.status(status).body(p);
}
```

```
PUT /claims/9f1c/contact-preferences
{ "version": 7, "contactMethod": "EMAIL", "phoneNumber": "abc", "emailAddress": "" }

400 Bad Request
Content-Type: application/problem+json
{ "status": 400, "title": "Request is invalid", "code": "INVALID_REQUEST",
  "detail": "2 fields are invalid", "traceId": "4bf92f35...",
  "errors": [
    { "field": "emailAddress", "code": "REQUIRED", "message": "Enter an email address" },
    { "field": "phoneNumber", "code": "FORMAT", "message": "Enter a phone number, like 01632 960
001" }
  ] }
```

```
// Express: one summary at the top, and the same message next to each field
const errorList = body.errors.map(e => ({ text: e.message, href: `#${e.field}` }));
```

Spring: `@Valid` runs Bean Validation on the request record; overriding `handleMethodArgumentNotValid` in the `ResponseEntityExceptionHandler` subclass shapes every `400`.

Why: a field error is something the user can fix on the page; a business-rule refusal (`422`, rule 26) is not, and Express shows the two differently. Using two status codes for that is a choice, made once; Express relies on it.

36. Run a bulk action as one request per record from Express; add a batch endpoint only when the business names the batch as one action, and then return a result per item

A team lead ticks 40 claims and presses **Close selected**.

Bad — one request, one transaction, all or nothing:

```
POST /claims/close-many
{ "claimIds": ["9f1c", "a201", ...38 more] }

422 Unprocessable Content
{ "code": "CLAIM_HAS_OPEN_ACTIONS", "detail": "Claim a201 has 1 note that needs action" }
```

One claim with an open note stops all 40, nobody can tell which 39 were fine, and the 40 version checks are gone.

Good — Express sends one close per claim, with that claim's version, in small batches, and shows a result per row:

```
const results = [];
for (const batch of chunk(selected, 5)) {
  await Promise.all(batch.map(async ({ id, version }) => {
    try { await claims.post(`/claims/${id}/close`, { version }); results.push({ id, ok: true });
}
    catch (e) { results.push({ id, ok: false, code: e.body?.code }); }
  }));
}
```

If the business names "close the month's batch" as one action of its own, the batch becomes a record: `POST /claim-batches/b7e2/close` with the batch's `version` answers `202` with `Location: /claim-batches/b7e2` (rule 21), the microservice still closes one claim per transaction, and `GET /claim-batches/b7e2` returns `status` and a list of `{ claimId, status, code }`.

Why: each record keeps its own rules and its own version check, and the user sees exactly which ones did not close. A batch endpoint with a result per item would do the same; starting from Express is our choice, because 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

A claimant fills in five pages of a new claim and presses **Save and come back later**.

Bad — the answers stay in the Express session:

```
req.session.newClaim = { ...req.session.newClaim, ...req.body };   // gone when the session ends
```

The session expires overnight, the claimant comes back, and every answer is lost. The Express session (rule 32) is for one sitting only.

Good — a draft record in the claim microservice, saved with `PUT` like any group of details, and no rule checked until the real create:

```
POST /claim-drafts
Idempotency-Key: 7d1e...
{ "claimantId": "c88" }

201 Created
Location: /claim-drafts/d420
{ "draftId": "d420" }

PUT /claim-drafts/d420/contact-preferences
{ "version": 2, "contactMethod": "EMAIL", "phoneNumber": "", "emailAddress": "jane@example.com" }

204 No Content

POST /claims
Idempotency-Key: 9ab0...
{ "fromDraftId": "d420" }                      // the rules run here, once

201 Created
Location: /claims/9f1c
{ "claimId": "9f1c" }
```

Why: a draft is allowed to be incomplete and wrong; a claim is not, and the one place that turns one into the other 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

Two caseworkers edit the contact details of claim `9f1c`. The second save gets `409`.

Bad — Express fixes it quietly:

```
catch (e) {
  if (e.status === 409) {
    const fresh = await claims.get(`/claims/${id}`);
    await claims.put(`/claims/${id}/contact-preferences`, { ...body, version: fresh.version });
// overwrites
  }
}
```

The first caseworker's new phone number is overwritten and nobody knows.

Good — a conflict page that says what happened and shows the current values:

```
catch (e) {
  if (e.status === 409) {
    const fresh = await claims.get(`/claims/${id}`);
    return res.render('conflict', {
      message: 'Someone else changed this claim while you were editing it.',
      current: fresh, yours: body,
      startAgain: `/claims/${id}/contact-details`,
    });
  }
  throw e;
}
```

Why: only the user can decide whose change should win, and `409` exists to ask them.

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

The payment microservice's listener for `ClaimClosedV1` needs the claim's payment schedule from `GET /claims/9f1c/payment-schedule`.

Bad:

```
http.get().uri("/claims/{id}/payment-schedule", claimId.value())
    .header("X-Api-Key", "shared-secret-2019")              // one key for every service, since
2019
    .retrieve().body(PaymentSchedule.class);
```

The key is in three repositories and a chat message, cannot be rotated without a coordinated release, and gives the holder every endpoint.

Good — the microservice signs in as itself and gets a short-lived token with only the permission it needs:

```
@Bean RestClient claimsClient(RestClient.Builder builder, ClientRegistrationRepository
registrations, OAuth2AuthorizedClientService clients) {
    var oauth = new OAuth2ClientHttpRequestInterceptor(new
AuthorizedClientServiceOAuth2AuthorizedClientManager(registrations, clients));
    oauth.setClientRegistrationIdResolver(request -> "claims");          // fetches, caches and
renews the token
    return builder.baseUrl(claimsUrl).requestInterceptor(oauth).build();
}

// application.properties: the service's own credentials
// spring.security.oauth2.client.registration.claims.client-id=payment-service
// spring.security.oauth2.client.registration.claims.client-secret=${CLAIMS_CLIENT_SECRET}
// spring.security.oauth2.client.registration.claims.authorization-grant-type=client_credentials
// spring.security.oauth2.client.provider.claims.token-
uri=https://gateway.example.gov.uk/oauth2/token
```

The gateway gives `payment-service` a token whose `permissions` holds only `VIEW_PAYMENT_SCHEDULE`, and the claim microservice checks it on that endpoint like any other permission (rule 15). A user's token is never stored for later use: it belongs to that user's request.

Spring: `OAuth2ClientHttpRequestInterceptor` with `AuthorizedClientServiceOAuth2AuthorizedClientManager` adds `Authorization: Bearer` to every request from any thread, a listener's included; Boot creates no manager bean, so the `@Bean` makes one. Only a `RestClient` from Boot's injected `RestClient.Builder` sends `traceparent` on.

Why: every caller is named, every call is limited to what that caller needs, and a leaked token dies on its own.

40. Receive an outside system's callback on its own endpoint: check the signature, store it with its ID, answer `202`, and raise our own event; no business logic in the handler

The document-check provider posts the result of a payslip check.

Bad — the callback handler does the work:

```
@PostMapping("/document-checks/callback")
public void callback(@RequestBody DocumentCheckCallback body) {
    Claim claim = claims.findById(body.claimReference()).orElseThrow();
    claim.verifyEvidence(body.documentId(), body.result());   // their IDs and words, straight in
    claims.save(claim);
}
```

Anyone who finds the URL can verify evidence; a slow save makes the provider time out and resend, and the second call fails on the already-verified item.

Good:

```
@RestController
class DocumentCheckCallbackController {
    @PostMapping("/document-checks/callback")                          // permitAll in the
security config: the signature is the check
    ResponseEntity<Void> callback(@RequestBody String rawBody, @RequestHeader("X-Signature")
String signature) {
        signatures.require(rawBody, signature);                        // 401 if wrong
        callbackRecorder.record(parse(rawBody), rawBody);
        return ResponseEntity.accepted().build();                      // 202: the work happens
from the event
    }
}

@Service
class CallbackRecorder {
    @Transactional
    public void record(DocumentCheckCallback body, String rawBody) {
        if (mongo.exists(query(where("_id").is(body.callbackId())), "callbacks")) return;   // a
resend: fine
        mongo.insert(new StoredCallback(body.callbackId(), rawBody), "callbacks");         //
unique _id
        mongo.insert(OutboxEntry.of(new DocumentCheckedV1(body.callbackId(), body.documentId(),
body.result())), "outbox");
    }
}
```

Spring: `@Transactional` with `MongoTransactionManager` stores the callback and its outbox entry together; `ResponseEntity.accepted()` is the `202`. A `SecurityFilterChain` bean with

`requestMatchers("/document-checks/callback").permitAll()` opens this one path; every other path keeps Boot's default, a valid token.

A listener on `DocumentCheckedV1` translates their document ID into our evidence ID and calls `claimService.verifyEvidence`.

Why: the provider gets a fast, repeatable answer, and the business rule runs where it always runs.

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

A team lead wants the open claims list as a spreadsheet.

Bad — Express builds the CSV from the list pages:

```
let rows = [];
for (let page = 1; page < 500; page++) rows.push(...(await claims.get(`/claims?
status=OPEN&page=${page}`)).items);
res.send(toCsv(rows));          // 500 calls, every column, no limit
```

The export takes minutes, pulls every field including the National Insurance number, and times out at the gateway.

Good — one endpoint next to the list, same filters, streamed, limited:

```
GET /claims/export?status=OPEN&sort=createdAt,desc
Accept: text/csv

200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="claims-open-2026-10-31.csv"
claimId,status,claimType,createdAt
9f1c,OPEN,STANDARD,2026-10-01T09:15:00Z
...                                        (at most 10,000 rows; 422 EXPORT_TOO_LARGE above that)
```

Columns with personal data appear only when the token has `EXPORT_CLAIM_PERSONAL_DATA`. An export the user cannot wait for answers `202` with a status URL (rule 21), and the file arrives at a download link later.

Why: an export is a read with the same rules as the list it comes from, plus a size the server can promise.

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

The claim page shows the claimant's name, which the eligibility microservice owns.

Bad — the claim microservice keeps a copy for display:

```
GET /claims/9f1c
{ "claimId": "9f1c", "claimantName": "Jane Doe", ... }   // copied when the claim was created
```

The claimant changes their name in eligibility; the claim page shows the old one for years, and two teams argue about which is right.

Good — Express reads the claim, then the claimant from its owner, and composes; the name is not essential, so the page still shows without it (rule 31):

```
const claim = await claims.get(`/claims/${id}`);                                  // essential
const claimant = await eligibility.get(`/claimants/${claim.claimantId}`).catch(() => null);
res.render('claim', { claim, claimantName: claimant?.fullName ?? 'Name not available' });
```

The claim microservice stores the eligibility *decision* (its rules need it) and the claimant *ID*, nothing else about the claimant.

Why: one owner per fact means one place to correct it.

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

A caseworker created claim `9f1c` for the wrong claimant.

Bad:

```
DELETE /claims/9f1c

204 No Content
```

The audit trail now points at a claim that does not exist, the `ClaimOpenedV1` event the payment microservice received has no matching close, and nobody can answer "why was this created?".

Good:

```
POST /claims/9f1c/cancel
{ "version": 3, "reason": "WRONG_CLAIMANT", "note": "Created against the wrong record" }

204 No Content

GET /claims?status=OPEN&sort=createdAt,desc&page=0&size=20     // cancelled claims are not in the
default list
GET /claims?status=CANCELLED&sort=createdAt,desc&page=0&size=20 // but can be listed
GET /claims/9f1c                                                // and still read
{ "claimId": "9f1c", "version": 4, "status": "CANCELLED", "cancelledOn": "2026-10-31",
"cancellationReason": "WRONG_CLAIMANT" }
```

Why: business records are history; `cancel` records the mistake, `DELETE` erases it (rule 22).

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

Creating a claim needs the eligibility microservice, and it is down.

Bad — the microservice hides it:

```
POST /claims
Idempotency-Key: 7d3e...
{ "claimantId": "c88", "claimType": "STANDARD" }
```

```
500 Internal Server Error
{ "error": "Connection refused" }
```

Express shows a generic error page, the alert counts a bug, and nobody can tell the caseworker that nothing was saved.

Good:

```
POST /claims
Idempotency-Key: 7d3e...
{ "claimantId": "c88", "claimType": "STANDARD" }

503 Service Unavailable
Retry-After: 30
Content-Type: application/problem+json
{ "status": 503, "title": "A service we depend on is unavailable", "code":
"DEPENDENCY_UNAVAILABLE",
  "detail": "Eligibility check is unavailable; nothing was saved", "traceId": "4bf92f35..." }
```

```
// in ApiErrors (rule 26)
@ExceptionHandler(DependencyUnavailable.class)                  // thrown by an adapter when the
other system is down (rule 9)
ResponseEntity<ProblemDetail> unavailable(DependencyUnavailable e) {
    ProblemDetail p = problem(503, "A service we depend on is unavailable",
"DEPENDENCY_UNAVAILABLE",
                              e.system() + " is unavailable; nothing was saved");
    return ResponseEntity.status(503).header("Retry-After", "30").body(p);
}
```

```
if (e.status === 503) return res.render('try-again-later', { seconds: e.headers['retry-after']
});
```

Spring: an `@ExceptionHandler` in the shared advice returns a `ResponseEntity`, so it can set the `Retry-After` header.

Express never retries a change on its own (rule 6). For a read, Express shows the page with what loaded (rule 31).

Why: a convention. `503` tells the user the truth, "nothing was saved, try later", `500` tells them nothing, and alerts can tell an outage next door from a bug of ours. The circuit breaker (rule 30) is what protects eligibility, 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

The claim team renames `createdAt` to `openedAt` and updates `openapi.yaml`.

Bad — the contract is documentation nobody runs:

```
# openapi.yaml, updated by hand, checked by nobody
ClaimSummary:
  properties:
```

```
    openedAt: { type: string, format: date-time }
```

```
// Express, a month later, in production
const opened = formatDate(claim.createdAt);        // undefined → "Invalid Date" on every row
```

The file says `openedAt`, the microservice still sends `createdAt`, and nobody finds out until every row shows "Invalid Date".

Good — two tests, one on each side:

```
// claim microservice: every endpoint test validates the real response against openapi.yaml
@SpringBootTest
@AutoConfigureMockMvc
class ClaimContractTest {
    @Autowired MockMvc mvc;

    @Test
    @WithMockUser(authorities = "VIEW_CLAIM")
    void listMatchesContract() throws Exception {
        MvcResult r = mvc.perform(get("/claims?
status=OPEN&sort=createdAt,desc&page=0&size=20")).andReturn();
        contract.assertValid("GET /claims", r.getResponse());   // fails the build on an unlisted
or renamed field
    }
}
```

Spring: `@SpringBootTest` with `@AutoConfigureMockMvc` runs the real endpoint; the check against `openapi.yaml` is our own code over the response.

```
// Express: the client is tested against the examples recorded in openapi.yaml
test('renders a claim summary', async () => {
  const claim = examples.get('ClaimSummary');     // { claimId: "9f1c", openedAt: "2026-10-
01T09:15:00Z", ... }
  expect(renderRow(claim)).toContain('1 October 2026');
});
```

For an Express test that needs a whole microservice rather than one example, WireMock plays the microservice, with stubs built from the same `openapi.yaml` examples, so the two sides never test against different data.

A rename now fails the Express build on the day the contract changes, and rule 27's three-release path becomes the fix.

Why: a contract that is tested on both sides breaks a build, not a page.

Sources: GOV.UK Design System: Check answers, GDS API technical and data standards, GOV.UK Service Manual: progressive enhancement.