iJS International JavaScript Conference iJS New York · devmio New York Week · October 2026

Backend Boundaries for Frontend Developers

Route-scoped, client-scoped, domain-scoped: a map for the backend decisions frontend developers make every day.

Nick Daniel – En Dash · endash.us

n-url.us/ufyjif

Three tickets from the n-brew backlog

NB-418 `feature`

Add a tip field to checkout.

Where does this mutation live?

NB-421 `bug`

/account/orders redirects when logged out, but the orders query still runs.

Where do we check the session?

NB-430 `feature`

Members get 10% off. "Can we just do it in the checkout action?"

How much logic is too much for a route?

None of these tickets says "architecture." All three ask **where the logic should live**.

Nick Daniel

FOUNDER, EN DASH · RICHMOND, VIRGINIA

endash.us

01 · THE MAP

Three buckets

Sorted by why the code exists, not where it runs.

Sort by why it exists

Route-scoped

Exists because *this URL* exists.

LIVES IN	`loader`, `action`
DOES	Parses the request, shapes the response
DIES WHEN	The screen is deleted

Client-scoped

Exists because *this client app* exists.

LIVES IN	`*.server.ts`, middleware, a BFF
DOES	Sessions, joining APIs, caching
DIES WHEN	The web app is retired

Domain-scoped

Exists because *the business* exists.

LIVES IN	A package, a service, an API
DOES	Pricing, inventory, who may refund
DIES WHEN	The café closes

Client-scoped means this client app, not the browser. A BFF is client-scoped code on a server.

Three questions sort any line of code

1 **Delete this route. Does the logic still need to exist?**
No means route-scoped.

ROUTE

2 **Would a native app need this, exactly as written?**
Yes, for sessions or payload shape: client. Yes, because it's a rule: domain.

CLIENT DOMAIN

3 **The rule changes. How many files change?**
More than one means domain logic is drifting.

DRIFT

Sort the checkout action

n–brew is a React Router app: a **loader** runs on the server before render, an **action** handles the form post. Eleven statements from the checkout action as it ships today. Sort each one. Keys 1 2 3 answer.

routes/checkout.tsx · action()

```
const form = await
request.formData();
```

```
const tip = Number(form.get("tip") ??
0);
```

```
if (!Number.isFinite(tip) || tip < 0)
return data({ error: "Bad tip" },
400);
```

```
return
redirect(`/orders/${order.id}`);
```

lib/session.server.ts · requireUser()

```
const user = await
requireUser(request);
```

```
session.flash("toast",
"Order placed!");
```

domain/orders.ts · placeOrder()

```
const subtotal = cart.items .reduce((s, i) ⇒ s +
i.priceCents * i.qty, 0);
```

```
const discount = user.isMember ?
Math.round(subtotal * 0.1) : 0;
```

```
if (!(await inventory.available(cart.items)))
return data({ error: "Sold out" }, 409);
```

```
const order = await db.orders.insert({ userId:
user.id, items: cart.items, subtotal, discount,
tip });
```

```
await mailer.send(receiptEmail(user.email,
order));
```

11 right · 0 either way · 0 missed

Start over

Same idea for reads: a loader, x-rayed

routes/menu.tsx

```tsx
export async function loader({ request }: Route.LoaderArgs) {
  const url = new URL(request.url);
  const cat = url.searchParams.get("cat") ?? "all";
  const user = await getOptionalUser(request);
  const menu = await catalog.list({ category: cat });
  const favs = user ? await favorites.forUser(user.id) : [];
  return data(
    { items: menu.map(toMenuCard), favs, locale: user?.locale },
    { headers: { "Cache-Control": "private, max-age=60" } },
  );
}
```

ROUTE ROUTE ROUTE CLIENT DOMAIN DOMAIN ROUTE ROUTE ROUTE ROUTE ROUTE

Route owns the request and the response: URL, params, headers, status, the shape this screen renders.

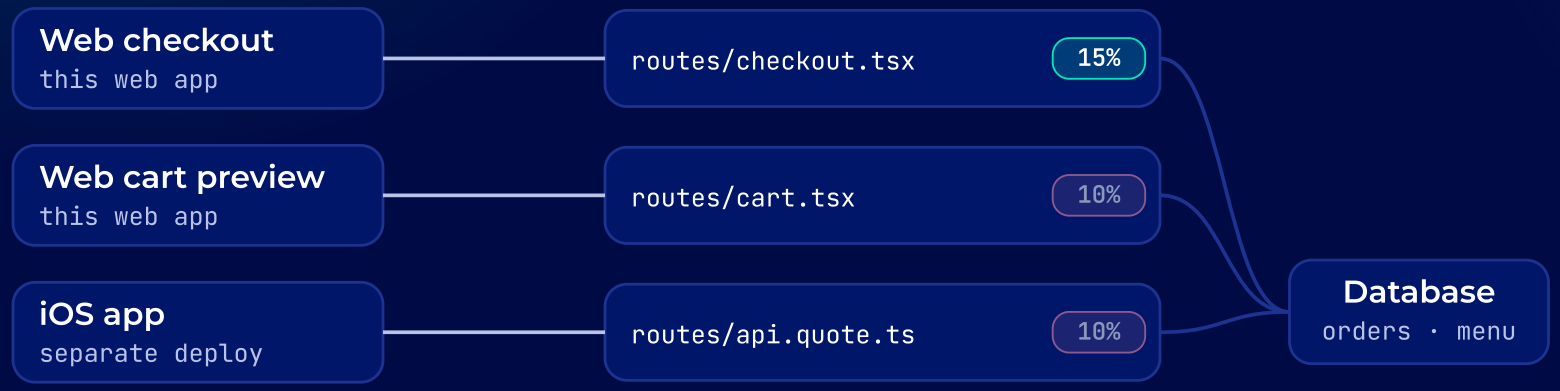
Client owns the session and anything every screen in this app shares.

Domain owns the facts: what is on the menu, what this user favorited.

The loader calls the other two buckets. It shouldn't contain them.

The second consumer

One rule: `members get 10% off`. Choose where it lives, add consumers, then change the rule.



RULE LIVES IN

Route | Web BFF | Domain

CONSUMERS

Web checkout

Web cart preview

iOS app

Admin refunds

Nightly report

COST

3 copies of the rule

1 hops for web checkout

2 of 3 copies still say 10%.

Rule changed in one place. 2 of 3 copies still say 10%.

This is what question 3 catches.

Change rule to 15%

Reset

02 · NB-421

Where do we check the session?

"Is there a user?" and "may this user do this?" are different questions in different buckets.

Two questions, two buckets, one redirect

Is there a user?

Cookie → session → user. Belongs to *this web client*. The iOS app has its own.

CLIENT · REQUIREUSER, MIDDLEWARE

May this user refund order 4411?

A business rule. Every consumer needs the same answer.

DOMAIN · ORDERS.REFUND() CHECKS POLICY

Where do they go when the answer is no?

`/login?next=…` or a 403. Up to this screen.

ROUTE · REDIRECT(), STATUS CODES

NB-421's redirect worked and the query still ran. The check itself was correct. It ran too late.

Where the check runs changes what runs

Six steps. Each one changes where the check runs, or what request comes in, and shows which loaders actually ran.

3 / 6 **Check in the layout loader**

← Back Next →

Only **account.tsx** checks now; children inherit its UI. Do they inherit the guard?

root.tsx RAN

routes/account.tsx REQUIREUSER THREW REDIRECT

routes/account.orders.tsx QUERY RAN

routes/admin.refunds.tsx REQUIREUSER DIDN'T RUN

WHAT ACTUALLY RAN

```
account loader: requireUser → redirect /login
```

```
orders loader ran in parallel: SELECT … WHERE
user_id = NULL
```

```
→ 302, but the query already happened.
```

In defense of "just put it in the route"

It's the right call when

- One consumer: this screen
- No rule a product manager would recognize
- Deleting the screen should delete the code

Five smells, in the order they usually arrive

- A **business rule** appears
- You want a **test without a Request**
- A **second consumer** shows up
- It must **run without a request**
- It **touches two aggregates**

One smell: note it in the PR. **Two:** extract this sprint. **Three:** you should have already.

03 · THE OTHER AXIS

Whose codebase is it?

The bucket says what the code is, not which repo or which team it belongs to.

Five homes

1 The browser

BUYS

Instant feedback.

COSTS

Anyone can bypass it.

ROUTE (UI)

2 The route file

BUYS

One PR.

COSTS

Invisible to other routes.

ROUTE

3 This app's server

BUYS

Shared by this app's screens.

COSTS

Other clients bypass it.

CLIENT

4 A domain package

BUYS

One rule. No hop.

COSTS

JS callers only, each on its own version.

DOMAIN

5 A service someone runs

BUYS

Any language, any client.

COSTS

A hop, and another team's roadmap.

DOMAIN

Move up a rung when the one you're on no longer holds.

When it should leave your codebase

Leave

- **Another team owns the rule.** Call their API.
- **Regulated or audited.** One place, one owner.
- **Consumers in other languages.**
- **Its own release cadence or secrets.**

Stay

- **One team owns the feature end to end.**
- **One deploy.** Web is the only consumer.
- **The rule is still changing.**

Middle path: a domain package in your repo. Move it out when one of the reasons on the left shows up.

The browser and the BFF *apply* decisions. Only the domain *makes* them.

Where should it live?

Describe the logic. Each home gets a verdict and a reason.

WHAT'S TRUE ABOUT THIS LOGIC?

- [x] Must hold even if the client is hostile
- [x] More than one screen needs it
- [ ] Another team owns the data or the rule
- [ ] Regulated or audited
- [ ] Consumers in other languages
- [x] One team owns the feature end to end

Preset: sort the menu

Preset: member discount

Preset: sales tax

TRAP

Browser

Can show the rule, can't enforce it.

TRAP

Route file

Two screens, two copies. Extract now.

ACCEPTABLE

This app's server

May call the rule. Must not own it.

RECOMMENDED

Domain package

One rule, one place, no hop.

ACCEPTABLE

Service someone runs

Not yet. A package gives the boundary without the hop.

The decision card

1. **Delete the route. Logic gone too? Route-scoped.**
2. **Would a native app need it as written?** Sessions or shape: **client**. A rule: **domain**.
3. **Authentication** is client. **Authorization** is domain. **The redirect** is route.
4. **Two smells, extract.** Rule, test, second consumer, no request, two aggregates.
5. **Start the domain as a package.** Make it a service when a second deploy, language, or team needs it.
6. **Browser and BFF apply decisions. The domain makes them.**

Let the tools ask the three questions

A PROMPT FOR WHATEVER ASSISTANT YOU USE

```
Review this route module. For each statement,
name its bucket:
  route-scoped   exists because this URL exists
  client-scoped  exists because this web app exists
  domain-scoped  exists because the business exists
Flag business rules, second consumers, and anything
that needs a Request to test. Propose the split.
```

OR MAKE IT A SCAN

- `src/domain` imports nothing from `react-router` or `node:http`. A lint rule.
- A route file over 40 lines, or one that mentions `user.is…`, gets a comment on the PR.
- n-dx SourceVision, the scanner you already have, or a 40-line script. Use whatever already runs on every PR.

Put it in the route.
Then know when to move it.

Route-scoped near the screen. Client-scoped in this app's server. Domain-scoped in a package or service somebody owns.

Go deeper

READ

- React Router docs: *Data Loading*, *Actions*, *Middleware*
- Sam Newman, *Pattern: Backends For Frontends*
- Vaughn Vernon, *Domain-Driven Design Distilled*
- Skelton & Pais, *Team Topologies*

TAKE HOME

- This deck and the demos: **endash.us/toolkit**
- The decision card as a one-pager
- The import-boundary lint config for `src/domain`

Before you go – three quick asks

Say hi

Questions after today, or a route handler you want a second opinion on.

`nick@endash.us`

Connect

LinkedIn and the rest of my links are on the site.

`endash.us`

Slides and demos

The deck, the demos, and the recording once it's posted.

`n-url.us/ufyjif`

We ask for
your feedback!

devmio