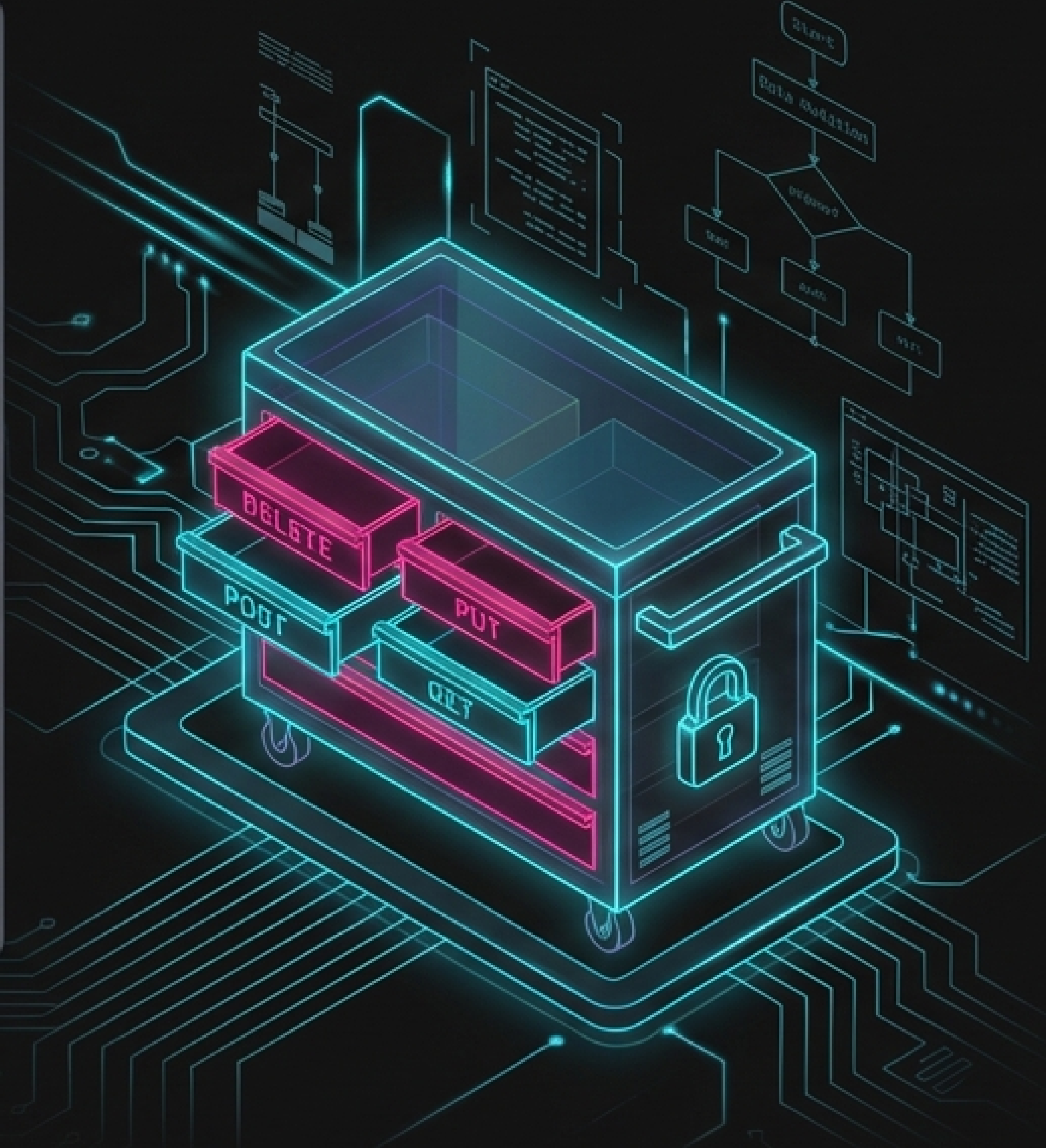
NODE/EXPRESS MUTATION LAB

Updating & Deleting Data

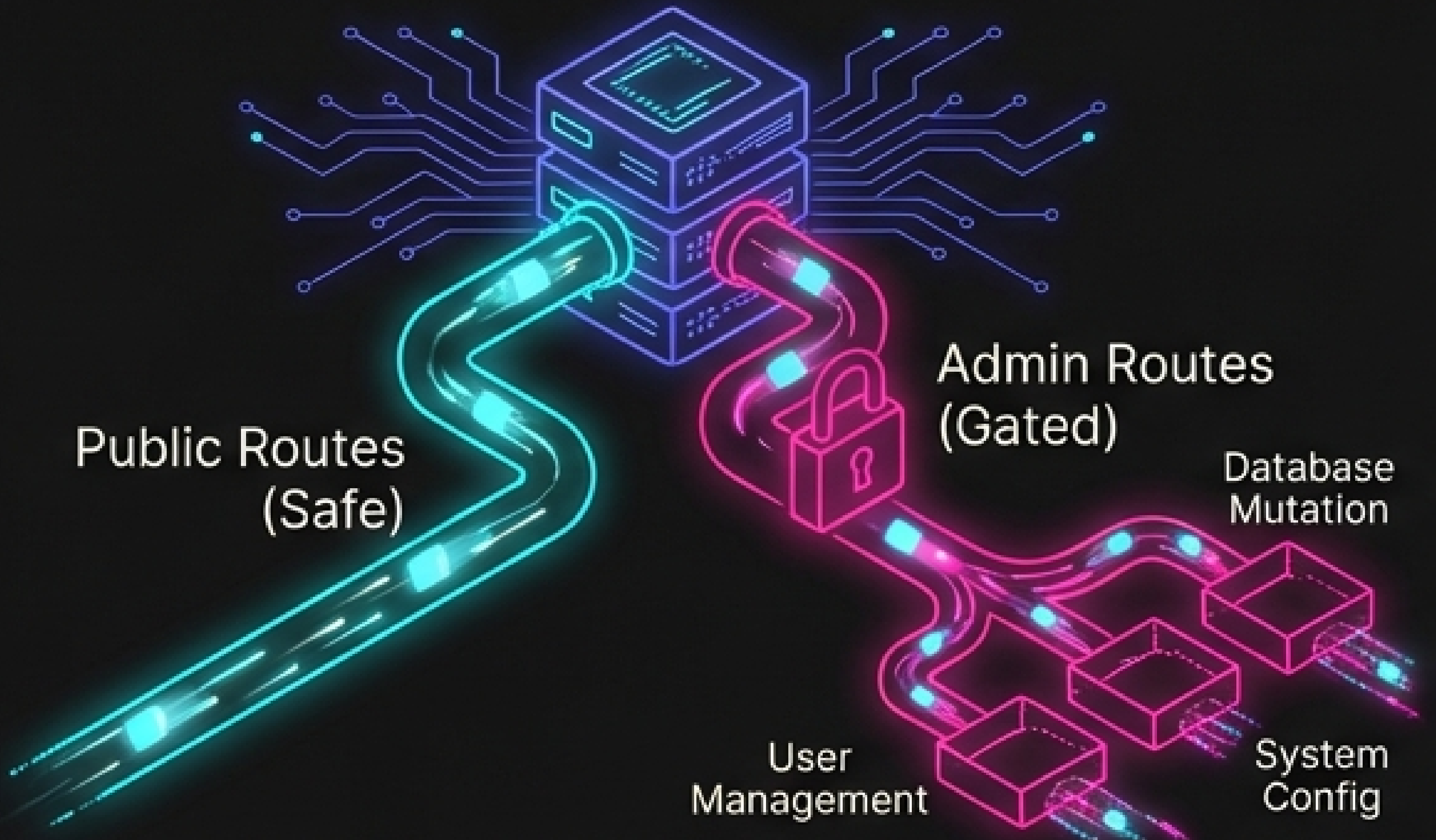
Moving beyond "Read-Only". We are entering the control room to build Admin Ops, Mongoose CRUD, and State Management.



```
> PROFESSOR SOLO: Welcome to the control room. Don't touch anything yet.
```

GATING THE GARAGE: THE ADMIN ROUTER

CONCEPTUAL DIAGRAM



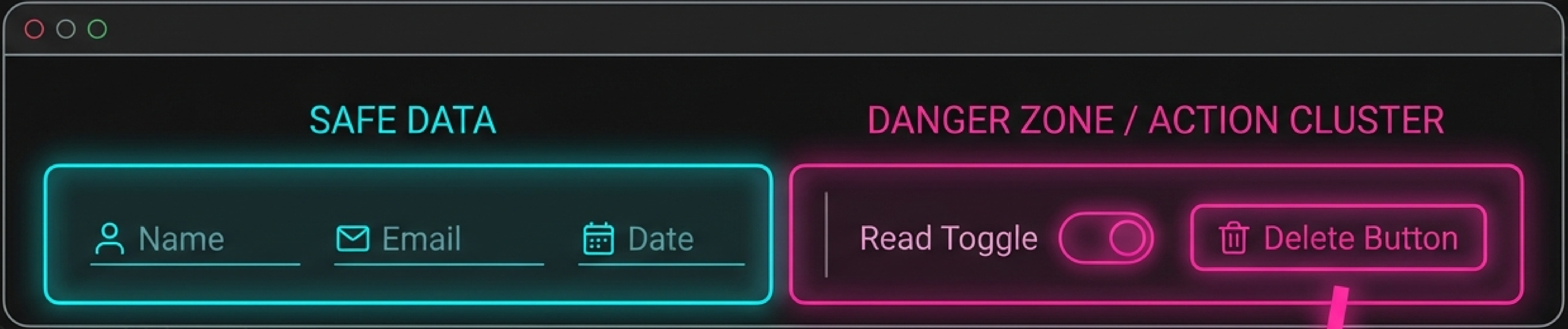
ADMIN ROUTER IMPLEMENTATION

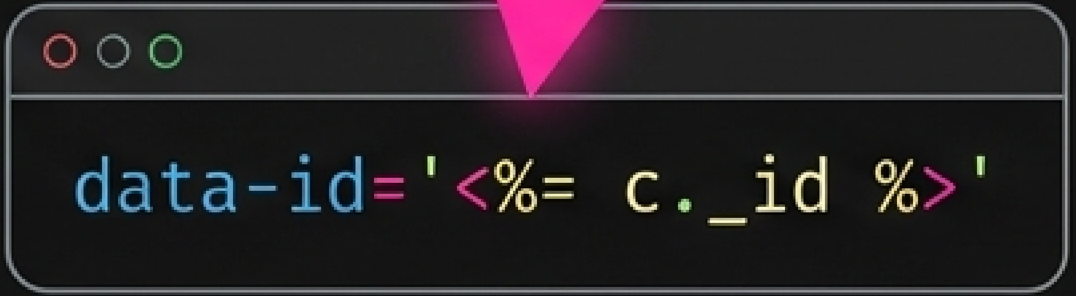
```
const express = require('express');
const router = express.Router();

// All routes here live under /admin/*
router.get('/contacts', async (req, res) => {
  // ... admin logic
});

module.exports = router;
```

Node2Know

Why /admin? It isolates mutation logic. Later, this creates a single choke-point for adding Authentication middleware. One lock, fifty routes secure.

THE CONTROL CONSOLE (LIST PATTERNS)

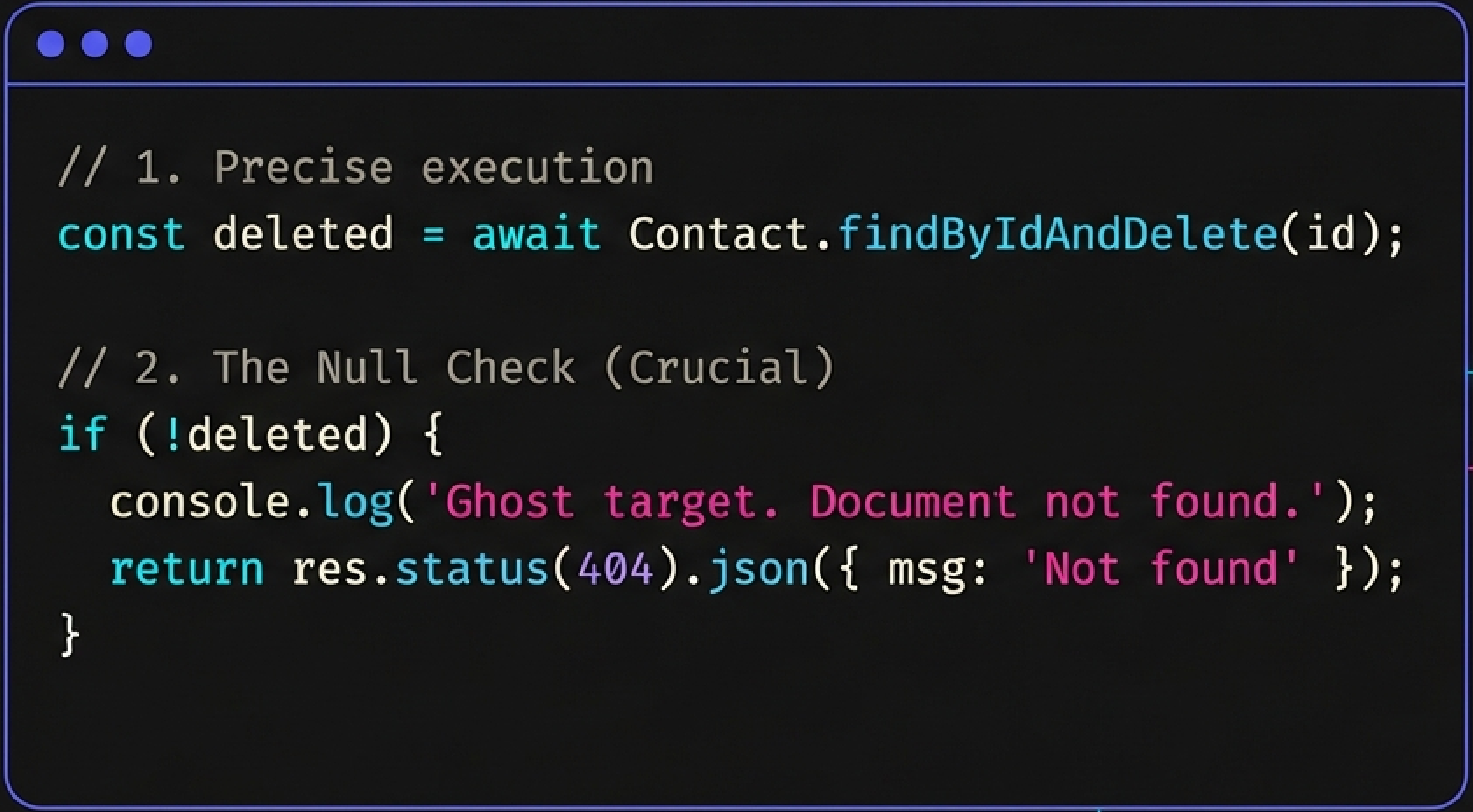




```
data-id='<%= c._id %>'
```

The ID is the detonator. Without it, the button is just decoration.

DELETE: THE BACK-END DEMOLITION



```javascript
// 1. Precise execution
const deleted = await Contact.findByIdAndDelete(id);

// 2. The Null Check (Crucial)
if (!deleted) {
  console.log('Ghost target. Document not found.');
  return res.status(404).json({ msg: 'Not found' });
}
```

METHOD: findByIdAndDelete(id). Precise and predictable.

Directly finds a document by its unique identifier and removes it in a single atomic operation.

THE GOTCHA: Mongoose does NOT crash if the ID isn't found; it returns null. You must handle this silent failure manually.

A common pitfall. Without the null check, your code might assume success even when nothing was deleted.

We return JSON here, not a redirect. The browser is listening via JS, not reloading.

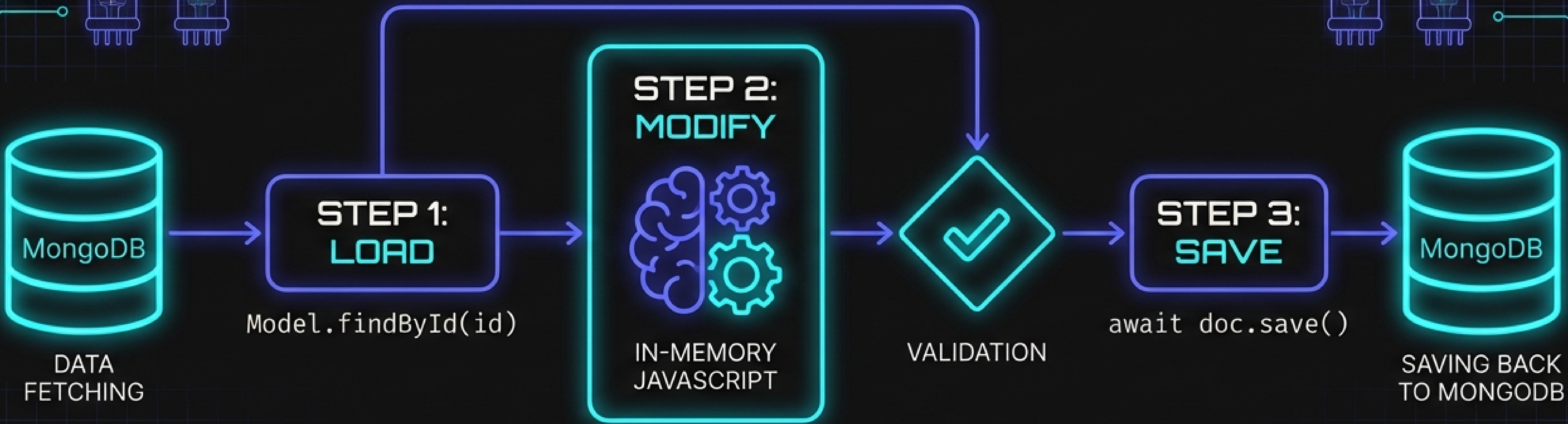
NotebookLM

DELETE: THE FRONT-END TRIGGER

- **Why Fetch?** HTML forms only support GET/POST. We need the DELETE verb.
- **Event Delegation**: One listener on the parent <ul> handles clicks for all <li> children.
- **UX Safety:** Always use confirm() before destruction.

```
list.addEventListener('click', async (e) => {
  // Event Delegation: Don't listen to 50
buttons. Listen to the container.
  const btn = e.target.closest('button.contact-
delete');
  if (!btn) return;

  const ok = confirm('Destroy this record?');
// The safety latch
  if(ok) {
    // ... fire fetch()
  }
});
```

UPDATE: THE LOAD → MODIFY → SAVE PATTERN



Why not `findByIdAndUpdate`? Direct updates often bypass schema rules. The 'Save' pattern ensures data integrity.

SIMPLE STATE CHANGE: THE TOGGLE
Implementing the Pattern

```
async toggleContactRead(id) {
  const contact = await Contact.findById(id); // LOAD
  if (!contact) return null;

  contact.isRead = !contact.isRead;           // MODIFY
  await contact.save();                       // SAVE
  return contact;
}
```

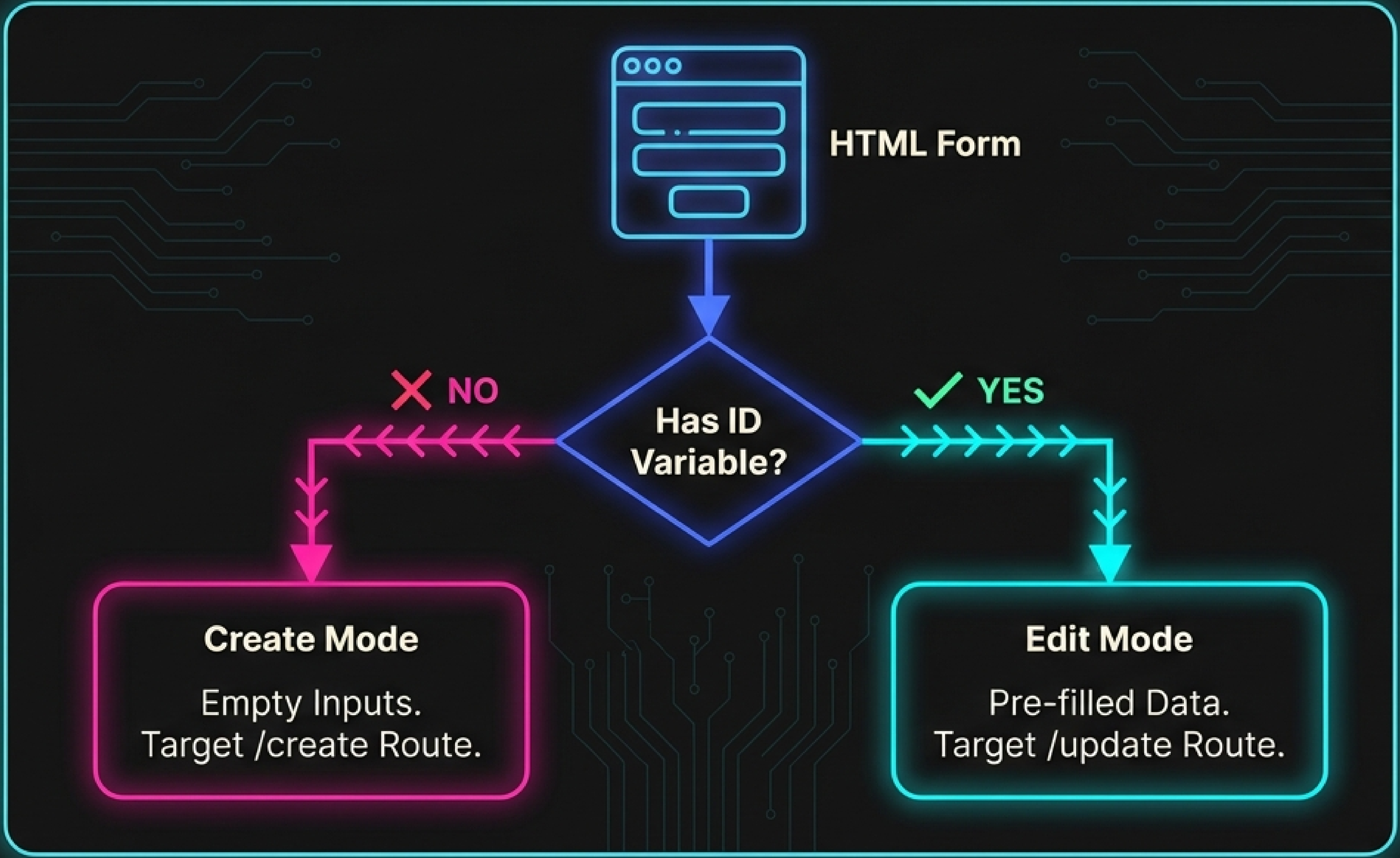
LOAD

MODIFY

SAVE

“This is a surgical strike. Load it, flip the switch, save it.”
— Professor Solo

COMPLEX UPDATES: THE DUAL-PURPOSE FORM



The DRY Principle:

create.ejs and edit.ejs are usually 99% identical. Don't maintain two files.

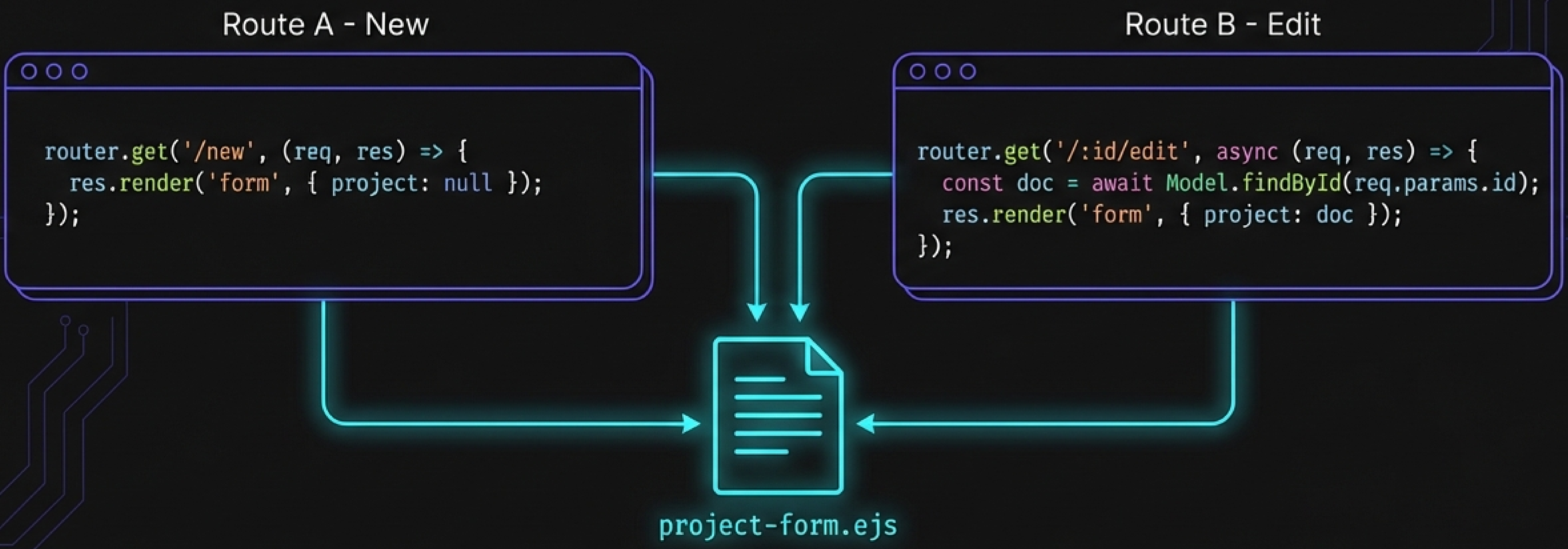
Don't roafleṅaion files. Use one file controlled by a signal.

THE EJS CONDITIONAL BOILERPLATE

```
<!-- Dynamic Action -->
<form action='<%= id ? `/update/${id}` : `/create` %>'>

  <!-- Safe Value Injection -->
  <input
    name='title'
    value='<%= project?.title || `` %>'
  />
```

CRITICAL: Always fallback to an empty string (**||** ''). If 'project' is **null** (Create Mode), EJS will crash the page without this safety net.

ROUTING TRAFFIC: ONE VIEW, TWO DESTINATIONS



Route A - New

```
router.get('/new', (req, res) => {
  res.render('form', { project: null });
});
```

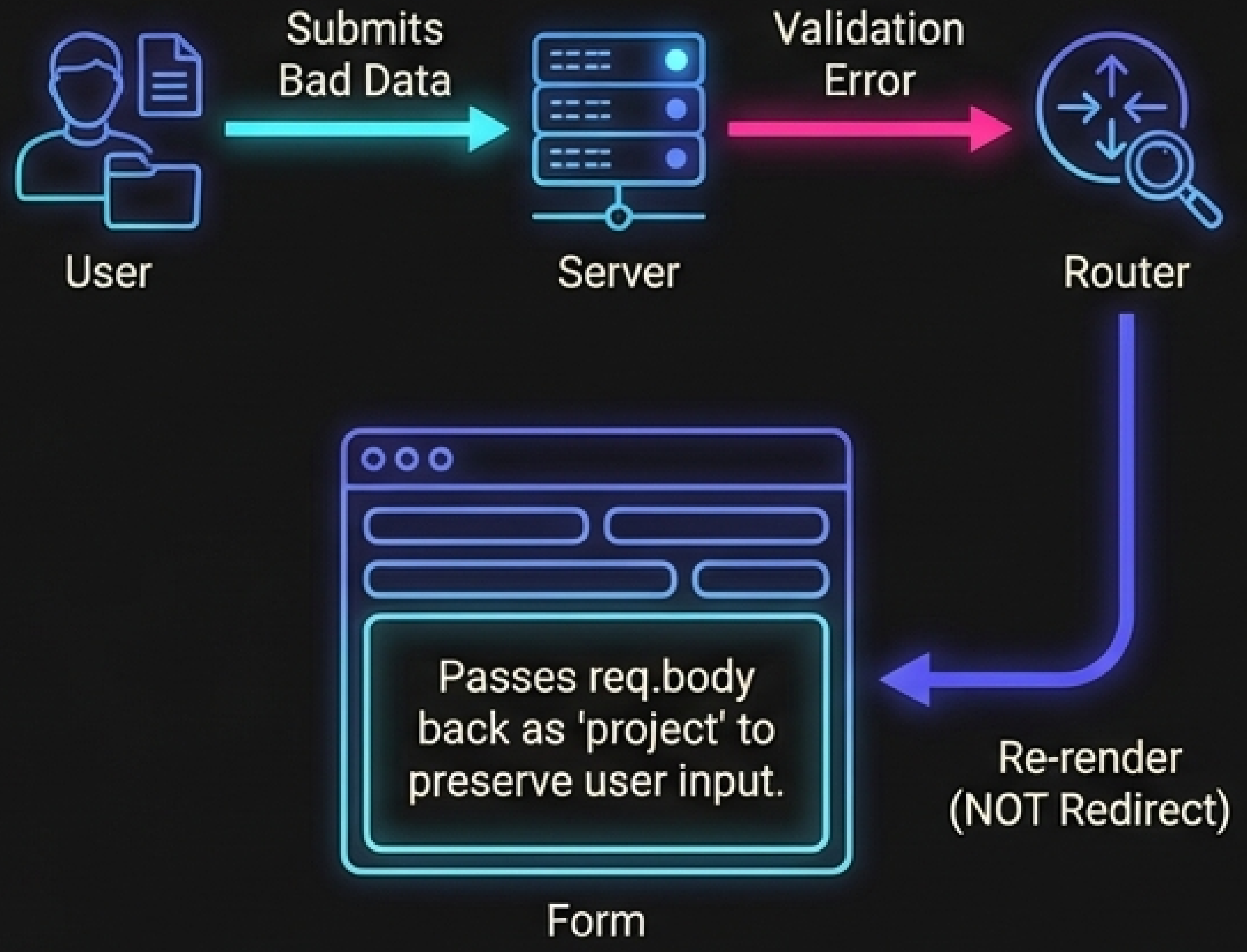
Route B - Edit

```
router.get('/:id/edit', async (req, res) => {
  const doc = await Model.findById(req.params.id);
  res.render('form', { project: doc });
});
```

The view adapts based on the context data it receives.

VALIDATION & THE "STICKY" FORM

ERROR HANDLING LOOP



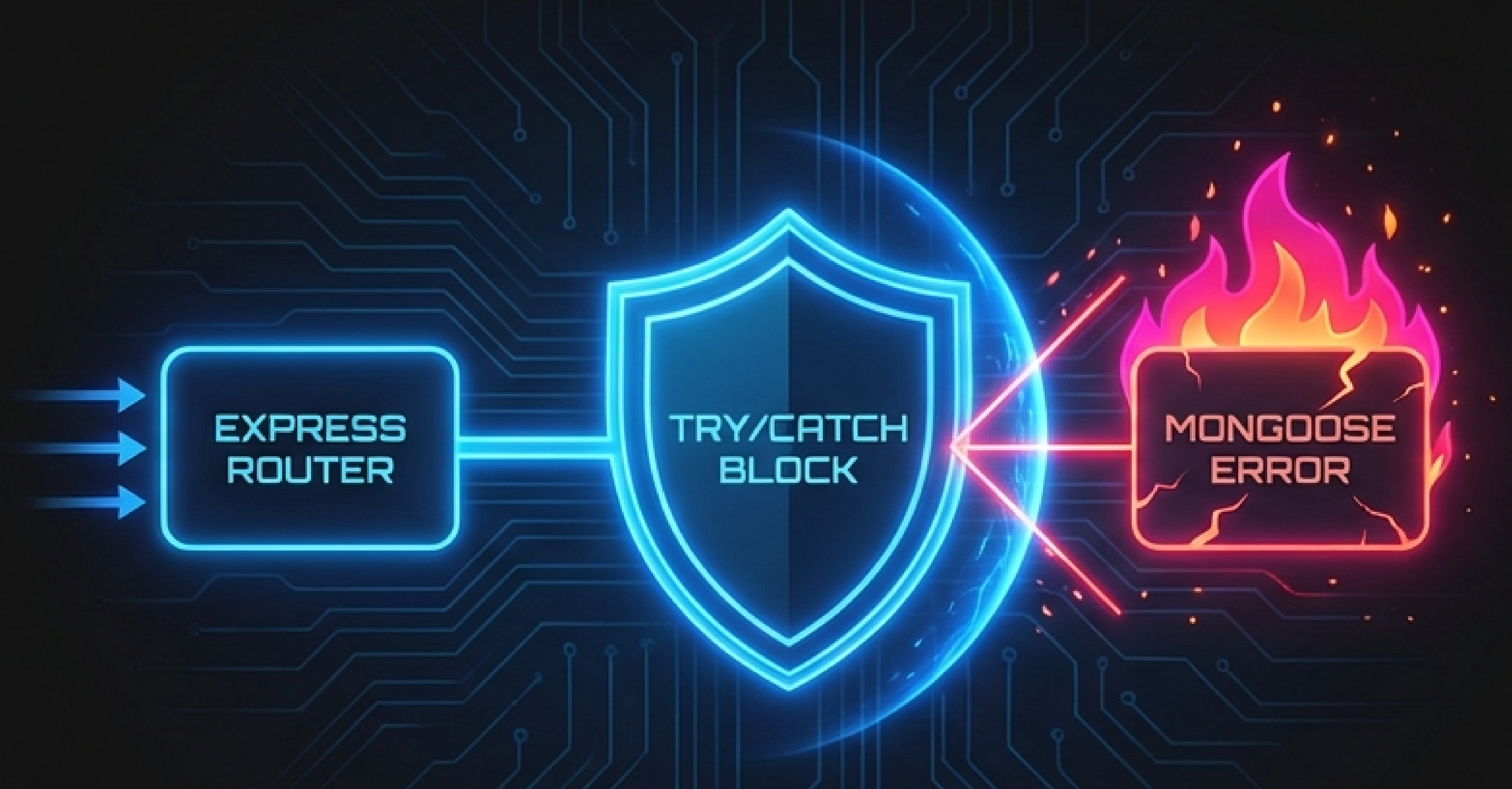
CATCH BLOCK IMPLEMENTATION

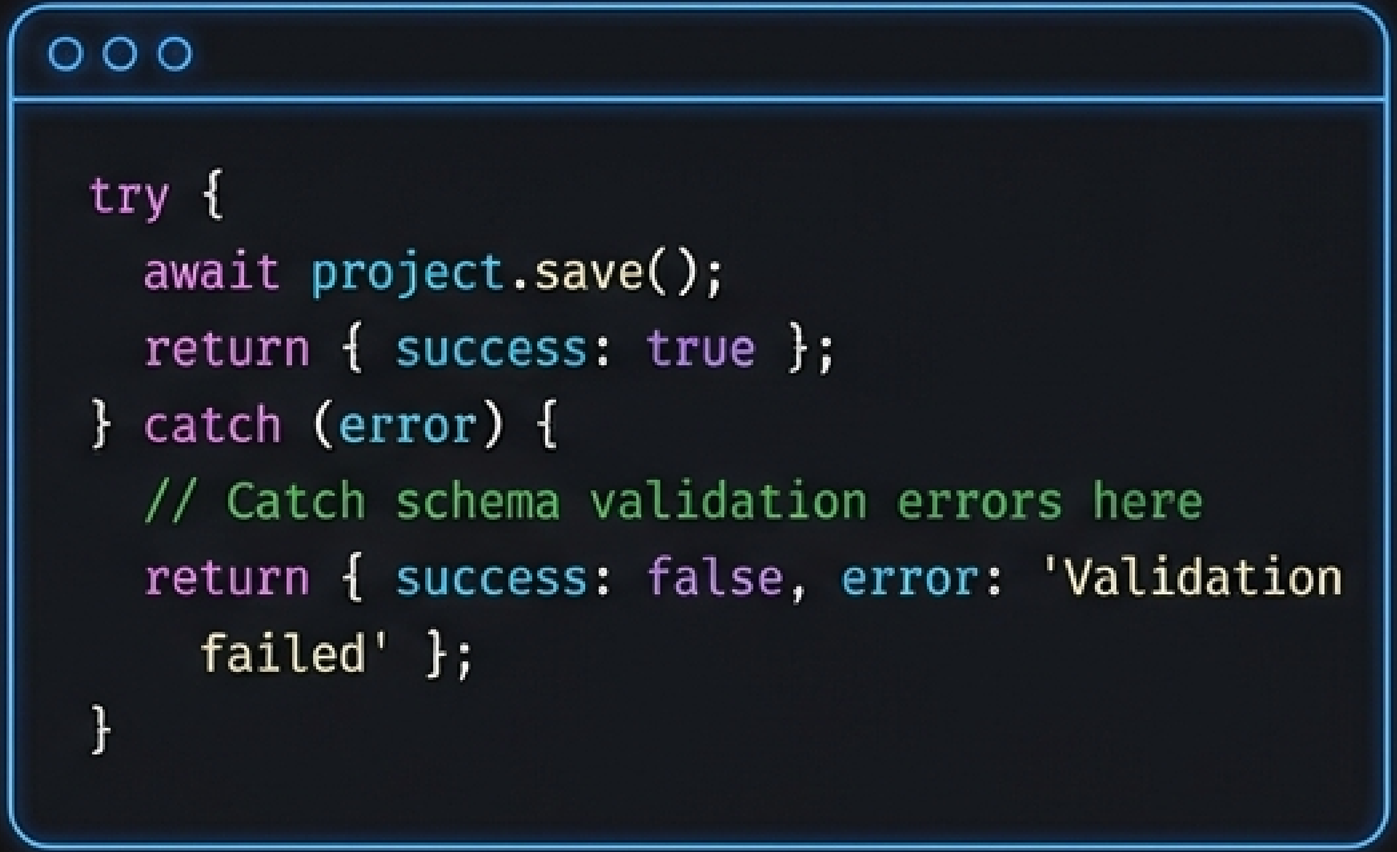
```
catch (err) {
  // Be kind to your users: don't
  make them re-type
  res.render('project-form', {
    project: req.body,
    error: 'Please fix the fields...'
  });
}
```

CRITICAL: This logic prevents data loss on validation failure.

NotebookLM

THE MONGOOSE OPS SHIELD





```
try {
  await project.save();
  return { success: true };
} catch (error) {
  // Catch schema validation errors here
  return { success: false, error: 'Validation
    failed' };
}
```

Embrace verbosity if it adds stability. Don't let a database error crash your entire server.

THE MUTATION MATRIX

	DELETE (The Scalpel)	UPDATE (The Truck)
Trigger	Client-side `fetch()`	Server-side `<form>`
Verb	`HTTP DELETE`	`HTTP POST`
Response	`JSON Data`	Redirect (or Re-render)
UX	DOM Removal (No Reload)	Full Page Navigation

Know when to use a scalpel and when to use a truck.

MUTATION PIT STOP: TROUBLESHOOTING

DIAGNOSTIC CHECKLIST

- ⚠ Did you 'await' the database call?
- ⚠ **Ghost IDs:** Did you handle 'null' when finding by ID?
- ⚠ **Form Action:** Is the URL pointing to the right ID (/contacts/<= id %>)?
- ⚠ **Event Defaults:** Did you 'e.preventDefault()' in your fetch listener?
- ⚠ **Method Limits:** HTML forms only support GET/POST. Are you using 'method=POST'?

THE MUTATION CODEX

MONGOOSE

```
Model.findByIdAndDelete(id)
doc.save() // Triggers validation
```

EXPRESS

```
res.json({ success: true })
res.redirect('/admin')
```

EXTRESS

```
Model.findByIdAndDlete(id)
doc.save() // Triggers validation
```

EJS

```
value='<%= project?.title || `` %>'
action='<%= id ? `/edit` : `/new` %>'
```

P.S. Keep learning. Security is next.