Phase 3 Training: Workers + Durable Objects + R2

What We Built

Three API endpoints on saltwaterbrc.com, each powered by a different Cloudflare developer platform product:

Build	Product	Endpoint	What It Does
1	Workers (Pages Functions)	`/api/status`	Returns site metadata + edge location
1	Workers (Pages Functions)	`/api/edge-info`	Request inspector — geo, TLS, bot signals
2	Durable Objects	`/api/visits`	Persistent visitor counter across all requests
3	R2 Storage	`/api/assets`	List, download, and upload files from edge storage

Live Endpoints

Try these right now in your browser — they return raw JSON, not a webpage:

- **saltwaterbrc.com/api/status** — Site metadata, stack info, and which Cloudflare data center served you
- **saltwaterbrc.com/api/edge-info** — Full request inspector: your geo location, TLS version, bot signals, network identity
- **saltwaterbrc.com/api/visits** — Durable Objects visitor counter (live after Build 2 deploy)
- **saltwaterbrc.com/api/assets** — R2 storage file browser (live after Build 3 deploy)

These are API endpoints — URLs that return data (JSON) instead of visual pages. This is how apps, websites, and services talk to each other. When your phone checks the weather, it hits an API endpoint. When a customer's site processes a payment, their frontend calls an API endpoint.

Workers vs Pages Functions — The Critical Distinction

This is one of the most important concepts to understand about the developer platform. **Pages Functions ARE Workers** — but with a simpler deployment model.

	Standalone Worker	Pages Function
What it is	Code deployed as its own project	Code dropped into `functions/` folder of a Pages project
How you deploy	`npx wrangler deploy` (manual)	`git push` (auto-deployed with your site)
How routes work	Manual route config in wrangler.toml	File path = route (`functions/api/status.js` → `/api/status`)
Where it lives	Separate project in CF dashboard	Part of your Pages project
Under the hood	A Worker	Also a Worker — Cloudflare compiles it into one
When to use	Needs Durable Objects, Cron Triggers, or advanced features	Stateless API logic alongside a website

Build 1: Cloudflare Workers (Pages Functions)

What It Is

Workers are serverless functions that run on Cloudflare's edge network — code executing in 300+ cities worldwide, responding in milliseconds. No origin server, no containers, no cold starts.

How We Did It

With **Pages Functions**, you don't need a separate Worker deployment. You create a `functions/` directory in your Pages project and files automatically become API endpoints:

```
site/ ■■■ functions/ ■ ■■■ api/ ■ ■■■ status.js → /api/status ■ ■■■ edge-info.js → /api/edge-info ■ ■■■
visits.js → /api/visits ■ ■■■ assets.js → /api/assets ■■■ index.html ■■■ wrangler.toml
```

Key Concept: File-Based Routing

- `functions/api/hello.js` → `/api/hello`
- `functions/api/todos/[id].js` → `/api/todos/:id` (dynamic)
- `functions/api/[[path]].js` → `/api/*` (catch-all)

The Code Pattern

Every Pages Function exports an `onRequest` handler:

```
export async function onRequest(context) { const { request, env } = context; // request = incoming HTTP request // env = bindings (R2, DO, KV, D1, etc.) // request.cf = edge metadata (geo, TLS, bot signals) return new Response("Hello from the edge!"); }
```

What request.cf Gives You (Free)

Every request to a Worker includes Cloudflare's edge metadata — no third-party APIs needed:

- **Geolocation**: country, city, region, latitude/longitude, timezone, continent, EU flag
- **Network**: ASN, organization (know if it's AWS, Google, a residential ISP, etc.)
- **Connection**: TLS version, cipher, HTTP protocol version
- **Bot signals**: bot score, verified bot flag, JA3 hash (TLS fingerprint)

Customer Pitch

"Every request that hits your domain already carries this intelligence — geolocation, network identity, bot signals. You're paying third-party providers for data that Cloudflare gives you for free on every Worker request."

Build 2: Durable Objects (Visitor Counter)

The Simple Explanation

A **Worker** is a program that runs when someone makes a request. It runs, responds, and forgets everything. It has no memory. If you asked it "how many people have visited this site?" it wouldn't know — because every time it runs, it starts fresh.

A **Durable Object** is a Worker that remembers. It has a notebook (storage) that persists between requests. You can write things down, and the next time someone asks, the information is still there.

But a Durable Object can't exist on its own. It needs a Worker to be its home. The Worker is the **building**, and the Durable Object is a **tenant** inside that building with their own private filing cabinet.

Here's how they work together in what we built:

1. **The counter Worker** (`saltwaterbrc-counter`) — this is the building. It hosts the `VisitorCounter` class. You deployed it with `npx wrangler deploy`. It's now running on Cloudflare's network.

2. **The VisitorCounter Durable Object** — this is the tenant. It lives inside that Worker. It has persistent storage where it keeps the visit count. Every time someone calls it, it reads the count, adds one, saves it, and returns the new number. It never forgets.

3. **The Pages Function** (`/api/visits`) — this is the front door on your website. When someone visits `saltwaterbrc.com/api/visits`, this function runs. But it can't count visits itself (it has no memory). So it reaches out to the Durable Object through the binding, says "add one to the count," gets the answer back, and returns it to the visitor.

The **binding** is the address book entry that tells the Pages Function where to find the Durable Object. That's what lives in `wrangler.toml` — the line that says "the thing called `VISITOR_COUNTER` is a `VisitorCounter` class living on the `saltwaterbrc-counter` Worker."

The Technical Details

Durable Objects provide **persistent, consistent state at the edge**. Each Durable Object instance:

- Has its own persistent storage (not a cache — real persistence)
- Is single-threaded (no race conditions)
- Is strongly consistent (reads always reflect the latest write)
- Is globally accessible but runs in one location for consistency

Why a Separate Worker?

Pages Functions **cannot host Durable Object classes directly**. The architecture is:

1. **Standalone Worker** (`counter-worker/`) — exports the `VisitorCounter` DO class 2. **Pages wrangler.toml** — binds to the DO via `script_name` 3. **Pages Function** (`/api/visits`) — calls the DO through the binding

```
# In the counter-worker wrangler.toml [durable_objects] bindings = [ { name = "VISITOR_COUNTER", class_name = "VisitorCounter" } ] [[migrations]] tag = "v1" new_classes = ["VisitorCounter"]
```

```
# In the Pages wrangler.toml [[durable_objects.bindings]] name = "VISITOR_COUNTER" class_name = "VisitorCounter" script_name = "saltwaterbrc-counter" # ← references the separate Worker
```

How the DO Stores State

```
// Write await this.state.storage.put("total", 42); // Read const total = await this.state.storage.get("total"); // List all keys const allKeys = await this.state.storage.list();
```

Storage is durable — data persists across requests, restarts, and deployments. It's not a cache. It's real storage.

idFromName: The Singleton Pattern

```
const id = env.VISITOR_COUNTER.idFromName("site-counter"); const stub = env.VISITOR_COUNTER.get(id);
```

Same name = same instance. Every request globally hits the same counter object. This is how you get consistency without a database.

Customer Pitch

- **Gaming**: "Epic Games manages millions of concurrent player sessions. Each session needs real-time state — health, inventory, position. Durable Objects give each session its own persistent, consistent state at the edge."
- **Fintech**: "Citco processes fund administration globally. Each transaction needs atomic, consistent state. Durable Objects provide single-threaded execution with strong consistency — no race conditions."
- **Real-time**: "Any app with collaborative features — shared documents, live dashboards, multiplayer — needs state that's fast and consistent. That's Durable Objects."

Build 3: R2 Storage (Asset Server)

What It Is

R2 is **S3-compatible object storage with zero egress fees**. Same API as S3, but you don't pay to read your own data. Ever.

The Numbers

Provider	Storage (1 TB)	Egress (1 TB/month)	Total
AWS S3	$23/mo	$90/mo	$113/mo
Google Cloud	$20/mo	$120/mo	$140/mo
Cloudflare R2	$15/mo	**$0/mo**	**$15/mo**

How It Works with Pages

R2 binds directly to Pages Functions through wrangler.toml:

```
[[r2_buckets]] binding = "ASSETS" bucket_name = "saltwaterbrc-assets"
```

Then in your function:

```
// List objects const listed = await env.ASSETS.list(); // Get an object const object = await
env.ASSETS.get("images/hero.png"); // Put an object await env.ASSETS.put("images/hero.png", fileBody, { httpMetadata:
{ contentType: "image/png" } });
```

Create the Bucket

```
wrangler r2 bucket create saltwaterbrc-assets
```

Customer Pitch

- **Any AWS customer**: "What are you paying in egress fees? R2 is S3-compatible — same API, same SDKs — with zero egress. Switch your bucket and stop paying to read your own data."
- **Healthcare**: "Medical imaging stored in R2, served through Workers. HIPAA-eligible, globally distributed, zero egress fees. No more paying AWS $90/TB just to serve images to your clinicians."
- **Media/Content**: "Every CDN request that pulls from your S3 bucket costs you money. R2 eliminates that entirely."

Deployment Steps

Build 1 (Workers / Pages Functions)

```
# Already deployed — Pages auto-detects the functions/ directory cd ~/Documents/Claude/SaltWaterBRC/site git add -A &&
git commit -m "Add Workers API endpoints" && git push # Auto-deploys via GitHub → Cloudflare Pages
```

Build 2 (Durable Objects)

```
# Step 1: Deploy the counter Worker cd ~/Documents/Claude/SaltWaterBRC/counter-worker npm install npx wrangler deploy
# Step 2: Push the Pages update (binding + API function) cd ~/Documents/Claude/SaltWaterBRC/site git push
```

Build 3 (R2 Storage)

```
# Step 1: Create the R2 bucket npx wrangler r2 bucket create saltwaterbrc-assets # Step 2: Push the Pages update
(binding + API function) cd ~/Documents/Claude/SaltWaterBRC/site git push
```

File Structure After Phase 3

```
SaltWaterBRC/ ■■■ site/ # Pages project (GitHub auto-deploy) ■ ■■■ functions/ ■ ■ ■■■ api/ ■ ■ ■■■ status.js #
Build 1: Site metadata ■ ■ ■■■ edge-info.js # Build 1: Request inspector ■ ■■■ blog/ ■ ■ ■■■
why-im-building-on-cloudflare.html ■ ■ ■■■ the-roadmap.html ■ ■■■ downloads/ # Training doc PDFs ■ ■ ■■■
```

```
DNS-Setup.pdf ■ ■ ■■■ Cloudflare-Pages-Deploy.pdf ■ ■ ■■■ Pages-vs-Workers-Setup.pdf ■ ■ ■■■
GitHub-to-Pages-Pipeline.pdf ■ ■ ■■■ Two-Repo-Sync-Setup.pdf ■ ■ ■■■ Phase3-Workers-DO-R2.pdf ■ ■ ■■■
Cloudflare-Sandbox-SDK.pdf ■ ■ ■■■ OpenCode-Overview.pdf ■ ■■■ index.html ■ ■■■ resources.html # Resources page
(PDF downloads) ■ ■■■ wrangler.toml # Pages config (bindings in dashboard) ■■■ counter-worker/ # Standalone Worker
(separate deploy) ■ ■■■ src/ ■ ■ ■■■ index.js # VisitorCounter DO + fetch handler ■ ■■■ wrangler.toml # Worker
config with DO migration ■ ■■■ package.json ■■■ Drafts/ # Blog post drafts ■■■ Training/ # Training docs (you're
reading one) ■■■ PROJECT-BRIEF.md # Master roadmap
```

Key Lessons

1. **Pages Functions = Workers without the separate deployment**. Drop a file in `functions/`, push to GitHub, done. 2. **Durable Objects need their own Worker**. Pages can bind to them but can't host the class. 3. **R2 binds directly to Pages** — no separate Worker needed. 4. **Every Worker request includes** `request.cf` — free geolocation, network intel, and bot signals on every request. 5. **Graceful degradation matters** — always handle the case where a binding isn't configured yet. Return a helpful error, not a crash.

Deployment Gotchas (Lessons from Real Errors)

These are mistakes we hit during deployment. Every one of them matters.

1. wrangler.toml MUST include `pages_build_output_dir`

Without this property, Pages sees the file as invalid and skips it. This caused "Unknown internal error" on every deploy until we added it.

```
pages_build_output_dir = "./"
```

2. Bindings go in the DASHBOARD, not wrangler.toml

For GitHub-connected Pages projects, `wrangler.toml` bindings don't get picked up reliably. We hit build failures every time the file referenced a DO binding. The fix: configure all bindings (Durable Objects, R2, KV, D1) in the Cloudflare dashboard under **Settings → Bindings**. The dashboard is the source of truth.

3. Pages Functions + Durable Object bindings = publish errors

Pages Functions that reference Durable Objects through `env.VISITOR_COUNTER` caused "Failed to publish your Function" errors at deploy time — even when the binding was correctly configured in the dashboard. The workaround: **call the standalone counter Worker directly from the frontend JavaScript** instead of through a Pages Function. Same Durable Object, same counter, different path.

```
// functions/api/visits.js — caused deploy errors const id = env.VISITOR_COUNTER.idFromName("site-counter"); const
stub = env.VISITOR_COUNTER.get(id);
```

```
// In HTML — calls the counter Worker's own URL fetch('https://saltwaterbrc-counter.saltwaterbrc.workers.dev/?page='
+ page)
```

4. Deploy bindings one at a time

When adding bindings in the dashboard, add them one at a time and test between each. We added DO and R2 separately to isolate which one caused issues. This saved hours of debugging.

5. R2 must be enabled in the dashboard first

Running `wrangler r2 bucket create` will fail with error code 10042 if R2 hasn't been enabled on the account. Go to **R2 Object Storage** in the dashboard and enable it before using the CLI.

6. Remove DO binding from dashboard if not using Pages Functions for DO

Since we call the counter Worker directly from the frontend, the DO binding on the Pages project isn't needed. Removing unnecessary bindings reduces deploy complexity and potential errors.

Final Architecture

```
Browser Request ■ ■■■ saltwaterbrc.com (Pages) ■ ■■■ Static HTML/CSS/JS (homepage, blog, resources) ■ ■■■
/downloads/*.pdf (training doc PDFs) ■ ■■■ /functions/api/ ■ ■■■ status.js → JSON (site metadata + edge info) ■
■■■ edge-info.js → JSON (request inspector) ■ ■■■ saltwaterbrc-counter.workers.dev (Standalone Worker) ■ ■■■
VisitorCounter Durable Object ■ ■■■ Persistent storage (visit counts) ■ ■■■ R2 Bucket: saltwaterbrc-assets ■■■
(Available for asset storage via /api/assets)
```

The site pages call the counter Worker directly via JavaScript. No Pages Function middleman needed for DO. Pages Functions work great for stateless Workers (status, edge-info). Durable Objects run best as standalone Workers called directly.