Cloudflare Durable Objects

Version Wrangler 4 · verified 2026-11-02

Single-threaded actors with private SQLite storage on Cloudflare — naming, storage, gates, alarms, WebSocket hibernation, limits, and class lifecycle.

Contents

Mental model 2

Classes, bindings, and exports 2

Addressing: names and IDs 5

SQLite storage 6

Input and output gates 8

Alarms 10

WebSocket hibernation 12

Location and limits 14

Class lifecycle and testing 15

Further reading 17

Mental model

A Durable Object is a single-threaded actor: one instance per name, worldwide, with its own private SQLite database on the same machine. You never lock anything — every request for “room-42” is routed to the one object that owns “room-42”, and that object handles them one at a time. Scale comes from having many objects, not from making one object faster, so the main design decision is what one object represents.

Classes, bindings, and exports

Piece	Purpose
`extends DurableObject`	Defines the class
`durable_objects.bindings`	Exposes it as `env.NAME`
`exports`	Provisions the class
`"storage": "sqlite"`	Backend for new classes
Public method	Callable over RPC

A Durable Object class lives in a Worker module and is reached through a binding on `env`. Public methods are

RPC endpoints: the caller gets a stub and `await` s them like local calls. The `exports` field in the Wrangler config declares each class; Cloudflare provisions a namespace on first deploy. It replaces the older `migrations` array, and a Worker uses one or the other, never both.

```
import {
  DurableObject,
} from "cloudflare:workers";

export class Room extends DurableObject {
  async post(user, text) {
    this.ctx.storage.sql.exec(
      "INSERT INTO msg VALUES (?, ?)",
      user, text,
    );
    return "ok";
  }
}
```

```
{
  "durable_objects": {
    "bindings": [
      { "name": "ROOM", "class_name": "Room" }
    ]
  },
  "exports": {
    "Room": {
      "type": "durable-object",
      "storage": "sqlite"
    }
  }
}
```

> **Gotcha:** A class exported in code but missing from `exports` is ignored — no namespace is provisioned, so the binding has nothing behind it. Declare every class.

Addressing: names and IDs

Call	Use
`env.ROOM.getByName(name)`	Stub for a named object
`env.ROOM.idFromName(name)`	Same ID, then `get(id)`
`env.ROOM.newUniqueId()`	Random ID; store it
`env.ROOM.idFromString(s)`	Rebuild a saved ID
`this.ctx.id.name`	Own name, inside the object

The same name always yields the same ID, so name objects after your unit of coordination: a room, a document, a tenant. Creating a stub does not start the object; the first method call does. Use `newUniqueId()` only when you will store the ID elsewhere, because you cannot find the object again without it.

```
export default {
  async fetch(req, env) {
    const url = new URL(req.url);
    const id = url.searchParams.get("room");
```

```
    const stub = env.ROOM.getByName(id);
    return Response.json(
      await stub.post("ada", "hi"),
    );
  },
};
```

> **Warning:** Always `await` stub calls. An un-awaited RPC call swallows its error and loses its return value.

SQLite storage

Call	Returns
`sql.exec(q, ...binds)`	Cursor
`cursor.toArray()`	All rows as objects
`cursor.one()`	Exactly one row, else throws
`storage.kv.get(key)`	Value, synchronously
`storage.transactionSync(fn)`	`fn` in one transaction
`storage.deleteAll()`	Wipes all storage

`this.ctx.storage.sql` is a synchronous SQLite API: there is no network hop, so queries return immediately. Create the schema in the constructor inside `blockConcurrencyWhile()`, which holds all other events until it finishes. `sql.exec()` cannot run `BEGIN` or `SAVEPOINT`; use `transactionSync()` instead. `PRAGMA user_version` is not supported, so track schema versions in a table of your own.

```
export class Room extends DurableObject {
  constructor(ctx, env) {
    super(ctx, env);
    ctx.blockConcurrencyWhile(async () => {
      ctx.storage.sql.exec(`CREATE TABLE IF NOT
        EXISTS msg (user TEXT, text TEXT)`);
    });
  }
  recent() {
    return this.ctx.storage.sql
      .exec("SELECT * FROM msg LIMIT 50")
```

```
      .toArray();  // [{ user, text }, ...]
  }
}
```

> **Gotcha:** A cursor held across an `await` has no snapshot isolation — it can return rows written after it was created. Call `.toArray()` before the next `await`.

Input and output gates

Mechanism	Guarantees
Input gate	No new event runs during storage I/O
Output gate	Replies wait until writes persist
Write coalescing	Writes without `await` commit atomically
`fetch()` or R2 call	Gate opens; others interleave

Single-threaded does not mean one request at a time: every `await` lets another event run. The runtime closes the gap for storage — while a storage call is pending, no other event is delivered, and a response is held until the writes it depends on are durable. Any other I/O, such as `fetch()`, opens the gate, so a check made before it may be stale after it.

```
async process(id) {
  const sql = this.ctx.storage.sql;
  const row = sql.exec(
    "SELECT status FROM job WHERE id = ?", id,
  ).one();
  if (row.status !== "pending") return;
  await fetch("https://api.example.com/run");
  // another request may have run here
  sql.exec(
    "UPDATE job SET status = 'done'"
      + " WHERE id = ? AND status = 'pending'",
    id,
```

```
  );
}
```

> **Tip:** Keep `blockConcurrencyWhile()` for initialization. It blocks every event; held across a `fetch()`, it caps the object's throughput at the speed of that call.

Alarms

Call	Effect
`storage.setAlarm(ms)`	Schedules `alarm()`; replaces any set
`storage.getAlarm()`	Time in ms, or `null`
`storage.deleteAlarm()`	Clears it
`alarm({ retryCount, isRetry })`	Runs at the time

Each object holds one alarm at a time. It is at-least-once: a throwing `alarm()` is retried with exponential backoff

starting at 2 seconds, up to 6 retries. Alarms do not repeat — set the next one inside the handler. For many timers, store them in a table and point the alarm at the earliest.

```
async alarm(info) {
  try {
    await this.flush();
  } finally {
    const next = Date.now() + 60_000;
    await this.ctx.storage.setAlarm(next);
  }
}
```

> **Gotcha:** After 6 failed retries the alarm stops until something calls `setAlarm()` again. Catch errors and reschedule yourself, as above, if the job must eventually run.

WebSocket hibernation

Call	Purpose
`ctx.acceptWebSocket(ws, tags?)`	Accept; allow hibernation
`webSocketMessage(ws, msg)`	Handler per message
`webSocketClose(ws, code, ...)`	Handler on close
`ws.serializeAttachment(v)`	Per-socket state, ≤ 16,384 bytes
`ctx.getWebSockets(tag?)`	Live sockets

With the Hibernation API the object can leave memory while clients stay connected; the next message re-runs the constructor and then the handler. Duration is not billed while it hibernates, and incoming messages are billed at a 20:1 ratio as requests. Keep per-connection data in attachments, because class fields are reset.

```
async fetch(req) {
  const pair = new WebSocketPair();
  const [client, server] = Object.values(pair);
  this.ctx.acceptWebSocket(server);
  server.serializeAttachment({ user: "ada" });
  return new Response(null, {
    status: 101, webSocket: client,
  });
}
async webSocketMessage(ws, msg) {
  const { user } = ws.deserializeAttachment();
  for (const s of this.ctx.getWebSockets()) {
    s.send(`${user}: ${msg}`);
  }
}
```

> **Gotcha:** Calling `server.accept()` instead of `this.ctx.acceptWebSocket(server)` works, but pins the object in memory and bills duration for as long as the socket is open.

Location and limits

Limit (SQLite backend)	Value
Storage per object	10 GB
Key plus value (KV API)	2 MB
CPU per request or message	30 s default, 5 min max
Throughput per object	~500–1,000 simple req/s
Classes per account	500 Paid / 100 Free

An object is created near the first request that reaches it and does not move afterwards. A `locationHint` (`wnam`, `weur`, `apac`, and others) on the first `get()` overrides that, best effort. A jurisdiction (`eu`, `us`, `fedramp`) is a hard guarantee of where the object runs and stores data.

```
const eu = env.ROOM.jurisdiction("eu");
const a = eu.getByName("tenant-7");
const b = env.ROOM.getByName("game-1", {
  locationHint: "weur",
});
```

> **Gotcha:** One global object is the classic bottleneck. A single “global” counter or rate limiter serializes all traffic through one thread — shard by user, room, or document.

Class lifecycle and testing

`exports` state	Effect
`"created"` (default)	Provisions the class
`"renamed"` + `renamed_to`	Moves data to a new class
`"deleted"`	Deletes the class and all data
`"transferred"`	Moves it to another Worker

Destructive changes are declared as tombstones in `exports` and applied on deploy. The storage backend is immutable: a namespace created as key-value storage cannot become SQLite, and switching it fails with `storage_type_mismatch`. Test with `@cloudflare/vitest-plugin`, which runs tests inside `workerd`; `runInDurableObject()` and `runDurableObjectAlarm()` come from `cloudflare:test`.

```
{
  "exports": {
    "OldRoom": {
      "type": "durable-object",
      "state": "renamed",
      "renamed_to": "Room"
    },
    "Room": {
      "type": "durable-object",
      "storage": "sqlite"
    }
```

```
  }
}
```

> **Warning:** A `deleted` tombstone permanently removes every object of that class and its data. There is no trash; copy what you need first.

Further reading

- Rules of Durable Objects
- SQLite-backed Storage API
- Durable Objects limits
- WebSockets and hibernation
- Durable Object class exports
- Testing Durable Objects