Cloudflare Workers

Version Wrangler 4 · verified 2026-10-29

Run JavaScript on Cloudflare's edge — the isolate model, bindings, KV, D1, R2, Durable Objects, Queues, Cron, limits, and safe deploys.

Contents

Mental model 2

Project and Wrangler CLI 2

The fetch handler 3

Configuration and compatibility 4

Variables, secrets, and types 6

Isolates, global scope, and waitUntil 7

KV: global, eventually consistent 8

D1: serverless SQLite 9

R2: object storage 11

Durable Objects: consistent state 12

Cron Triggers and Queues 14

Service bindings and RPC 15

Limits that shape your design 16

Environments, versions, and rollbacks 17

Further reading 18

Mental model

A Worker is a JavaScript module that Cloudflare runs in V8 isolates on its network, not in a container or VM you size. An isolate starts in milliseconds and serves many requests concurrently, so module-level variables are shared between unrelated requests and can vanish at any moment. Everything outside your code — storage, queues, secrets, other Workers — reaches you as a **binding** on the `env` object, declared in the Wrangler config, never as a URL plus an API key.

Project and Wrangler CLI

Command	Does
`npm create cloudflare@latest`	Scaffold a project
`npx wrangler dev`	Run locally on `workerd`
`npx wrangler deploy`	Upload and serve all traffic
`npx wrangler tail`	Stream production logs
`npx wrangler types`	Generate `Env` types

Wrangler is the CLI. It installs per project as a devDependency, so the whole team runs one pinned version. `wrangler dev` executes your code in a local `workerd` process and simulates every binding locally with Miniflare. `wrangler deploy` uploads a new version and sends 100% of traffic to it.

```
npm create cloudflare@latest -- my-worker
cd my-worker
```

```
npx wrangler dev      # http://localhost:8787
npx wrangler deploy   # *.workers.dev URL
```

> **Gotcha:** Local KV, D1, and R2 start empty — they are simulations, not your production data. Add `"remote": true` to a binding to use the real resource from `wrangler dev`.

The fetch handler

Parameter	Is
`request`	Standard `Request`
`env`	Bindings, vars, secrets
`ctx`	`waitUntil()`, `passThroughOnException()`

The default export is an object of handlers. `fetch` runs for every HTTP request and returns a `Response`. The APIs are Web standards — `URL`, `Headers`, `fetch`, `Response.json`, streams — and there is no built-in router: branch on the URL yourself or use a small framework such as Hono.

```
export default {
  async fetch(request, env, ctx) {
    const url = new URL(request.url);
    if (url.pathname === "/health") {
      return new Response("ok");
```

```
    }
    if (request.method === "POST") {
      const body = await request.json();
      return Response.json({ got: body });
    }
    return new Response("Not found", {
      status: 404,
    });
  },
};
```

> **Gotcha:** Bodies are streams and can be read only once. Reading `request.json()` and then forwarding the same `request` with `fetch` throws. Call `request.clone()` before the first read when you need the body twice.

Configuration and compatibility

Key	Purpose
`main`	Entry module
`compatibility_date`	Pins runtime behavior
`compatibility_flags`	Opt in or out per feature
`vars`	Plain-text variables
`routes`	Custom domain or route
`observability`	Turns on Workers Logs

Cloudflare recommends `wrangler.jsonc` over `wrangler.toml` for new projects; some newer features are JSON-only. The runtime updates continuously, but backwards-incompatible changes are gated behind dates: your Worker keeps the semantics of its `compatibility_date` until you bump it. Set it to today on new projects.

```
{
  "name": "my-worker",
  "main": "src/index.js",
  "compatibility_date": "2026-10-29",
  "observability": { "enabled": true },
  "vars": { "API_BASE": "https://api.example.com" },
  "routes": [
    {
      "pattern": "api.example.com",
      "custom_domain": true
    }
  ]
}
```

> **Gotcha:** Node.js built-ins such as `node:crypto` and `Buffer` need the `nodejs_compat` flag for compatibility dates from 2024-09-23 to 2026-08-03. From 2026-08-04 they are on by default. An npm package that fails to import `node:` modules usually means an old date without the flag.

Variables, secrets, and types

Source	Set with
`vars`	Wrangler config (committed)
Secret	`wrangler secret put NAME`
Local secret	`.dev.vars` or `.env` file
`Env` type	`wrangler types`

All of these land on `env` under the same name. Secrets are encrypted and never displayed again after upload; for `wrangler dev`, put them in a gitignored `.dev.vars` file. Do not hand-write an `Env` interface — `wrangler types` writes `worker-configuration.d.ts` from your config, so re-run it whenever a binding changes.

```
npx wrangler secret put STRIPE_KEY  # prompts
echo "STRIPE_KEY=sk_test_1" >> .dev.vars
npx wrangler types
```

> **Gotcha:** `wrangler secret put` creates a new version and deploys it immediately. During a gradual rollout, use `wrangler versions secret put` so the secret ships with the next version instead.

Isolates, global scope, and waitUntil

Do	Don't
Pass request data as arguments	Keep it in module variables
`ctx.waitUntil(promise)`	Leave promises un-awaited
Do I/O inside a handler	Call KV or `fetch` at top level

One isolate serves many requests and is evicted without warning, so treat module scope as a cache at best. I/O is tied to the request that started it: top-level KV calls fail, and reusing a body or stream from another request throws `Cannot perform I/O on behalf of a different request.` Work still running after you return a `Response` can be cancelled — hand it to `ctx.waitUntil()`, which allows up to 30 seconds after the response.

```
export default {
  async fetch(request, env, ctx) {
    const path = new URL(request.url).pathname;
    ctx.waitUntil(
      fetch("https://logs.example.com/hit", {
```

```
        method: "POST",
        body: JSON.stringify({ path }),
      }),
    );
    return Response.json({ path });
  },
};
```

> **Gotcha:** `const { waitUntil } = ctx` loses `this` and throws `Illegal invocation` at runtime. Always call it as `ctx.waitUntil(...)`.

KV: global, eventually consistent

Call	Returns
`env.KV.get(key)`	String or `null`
`env.KV.get(key, "json")`	Parsed value or `null`
`env.KV.put(key, value, opts)`	`Promise<void>`
`env.KV.delete(key)`	`Promise<void>`
`env.KV.list({ prefix })`	`{ keys, list_complete, cursor }`

KV suits read-heavy data that tolerates staleness: config, feature flags, cached renders. Reads are cached near the user, so a write can take 60 seconds or more to appear in other locations. Create a namespace with `npx wrangler kv namespace create FLAGS`, then add it under `kv_namespaces` with its `binding` and `id`.

```
await env.FLAGS.put(
  "beta",
  JSON.stringify({ on: true }),
  { expirationTtl: 3600 },
);
const beta = await env.FLAGS.get("beta", {
  type: "json",
  cacheTtl: 300,
});  // { on: true } or null
```

> **Gotcha:** Never use KV for counters or read-modify-write. Concurrent writes to one key overwrite each other, and more than one write per second to the same key returns `429`. Use a Durable Object when the value must be consistent.

D1: serverless SQLite

Method	Returns
`stmt.first()`	First row or `null`
`stmt.all()`	`{ results, meta }`
`stmt.run()`	`{ success, meta }`
`stmt.raw()`	Rows as value arrays
`env.DB.batch([...])`	Runs as one transaction

D1 is SQLite behind a binding. Always `prepare()` a statement and `bind()` values: binding prevents SQL injection and lets you reuse the

statement. `batch()` runs its statements as a single SQL transaction — if one fails, the whole sequence rolls back.

```
const user = await env.DB
  .prepare("SELECT * FROM users WHERE id = ?")
  .bind(id)
  .first();  // row object or null

const debit = env.DB.prepare(
  "UPDATE acct SET bal = bal - ? WHERE id = ?",
);
const credit = env.DB.prepare(
  "UPDATE acct SET bal = bal + ? WHERE id = ?",
);
await env.DB.batch([
  debit.bind(50, "a"),
  credit.bind(50, "b"),
]);
```

> **Gotcha:** `wrangler d1 execute` targets the local database unless you pass `--remote`. Forgetting it is why a migration “ran” but production still has no table.

R2: object storage

Call	Returns
`env.BUCKET.put(key, body)`	`R2Object`
`env.BUCKET.get(key)`	`R2ObjectBody` or `null`
`env.BUCKET.head(key)`	`R2Object` or `null`
`env.BUCKET.delete(key)`	`Promise<void>`
`env.BUCKET.list(opts)`	`{ objects, truncated, cursor }`

R2 is S3-compatible object storage, bound under `r2_buckets`. Stream bodies straight through — `request.body` into `put()`, `object.body` into a `Response` — so large files never have to fit in the 128 MB memory limit. Use `httpEtag` rather than `etag` in response headers: it is already quoted as HTTP requires.

```
const url = new URL(request.url);
const key = url.pathname.slice(1);
if (request.method === "PUT") {
  await env.BUCKET.put(key, request.body);
  return new Response("stored", { status: 201 });
}
const obj = await env.BUCKET.get(key);
if (obj === null) {
  return new Response("Not found", { status: 404 });
}
const headers = new Headers();
```

```
obj.writeHttpMetadata(headers);  // content-type…
headers.set("etag", obj.httpEtag);
return new Response(obj.body, { headers });
```

Durable Objects: consistent state

Piece	Purpose
`class X extends DurableObject`	Defines the object
`env.X.getByName(name)`	Stub for one instance
`await stub.method()`	RPC call into it
`this.ctx.storage`	Private SQLite storage
`exports` in config	Provisions the class

Each name maps to exactly one instance worldwide. It runs single-threaded and owns private, strongly consistent storage — the place for counters, rate limiters, chat rooms, and WebSockets. Public methods on the class are callable through the stub as RPC.

```
import { DurableObject } from "cloudflare:workers";
export class Counter extends DurableObject {
  async increment() {
    const n = await this.ctx.storage.get("n") ?? 0;
    await this.ctx.storage.put("n", n + 1);
    return n + 1;
  }
}
```

```
export default {
  async fetch(req, env) {
    const id = req.headers.get("x-user") ?? "anon";
    const stub = env.COUNTER.getByName(id);
    return Response.json(await stub.increment());
  },
};
```

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

> **Gotcha:** One object is one thread, good for roughly 500–1,000 simple requests per second. Sending all traffic to `getByName("global")` builds a global bottleneck — shard by user, room, or document instead.

Cron Triggers and Queues

Handler	Runs on
`fetch(request, env, ctx)`	HTTP request
`scheduled(controller, env, ctx)`	Cron Trigger, in UTC
`queue(batch, env, ctx)`	Batch of queue messages

One module can export several handlers. Crons live in `triggers.crons`, and `controller.cron` says which expression fired. A producer calls `env.JOBS.send(body)`; a consumer receives batches (10 messages by default) and calls `ack()` or `retry()` per message. Locally, trigger a cron with `curl localhost:8787/cdn-cgi/local/scheduled`.

```
export default {
  async scheduled(controller, env, ctx) {
    await env.JOBS.send({ task: controller.cron });
  },
  async queue(batch, env) {
    for (const msg of batch.messages) {
      try {
```

```
        console.log("job", msg.body);
        msg.ack();
      } catch {
        msg.retry({ delaySeconds: 60 });
      }
    }
  },
};
```

> **Gotcha:** Queues deliver at least once, so a message can arrive twice. Make consumers idempotent — key side effects on a message or job ID rather than assuming a single delivery.

Service bindings and RPC

Piece	Purpose
`"services"` in config	Binds another Worker
`extends WorkerEntrypoint`	Exposes methods as RPC
`await env.AUTH.check(t)`	Calls it, no HTTP

A service binding lets one of your Workers call another directly, without a trip over the public internet and at no extra cost. Methods on a `WorkerEntrypoint` class become RPC methods; the caller declares `{ "binding": "AUTH", "service": "auth-worker" }` under `services`.

```
// auth-worker/src/index.js
import {
  WorkerEntrypoint,
} from "cloudflare:workers";

export default class extends WorkerEntrypoint {
  async check(token) {
    return token === this.env.API_TOKEN;
  }
}
```

> **Note:** A new `WorkerEntrypoint` instance is created per call. It is still stateless — use a Durable Object to keep state between calls.

Limits that shape your design

Limit	Free	Paid
CPU time per request	10 ms	30 s default, 5 min max
Memory per isolate	128 MB	128 MB
Subrequests per request	50	10,000
Requests	100,000/day	No limit
Worker size (uncompressed)	64 MiB	64 MiB

CPU time counts only execution: waiting on `fetch`, KV, or D1 does not, so an I/O-bound Worker fits easily in 10 ms. Exceeding CPU or memory returns Error 1102, `Worker exceeded resource limits`. On the Paid

plan, raise the CPU ceiling in config, and stream large bodies instead of buffering them.

```
{
  "limits": { "cpu_ms": 300000 }
}
```

> **Gotcha:** Only six connections per request can wait for response headers at once. A seventh `fetch()`, KV, or R2 call waits for one of them, so a wide `Promise.all` fan-out runs in waves.

Environments, versions, and rollbacks

Command	Effect
`wrangler deploy --env staging`	Deploys `my-worker-staging`
`wrangler versions upload`	New version, no traffic
`wrangler versions deploy`	Splits traffic by percent
`wrangler rollback [id]`	Returns to an earlier version

Each environment under `env` in the config deploys as a separate Worker named `{name}-{env}`. For a gradual rollout, upload a version, then shift traffic in steps. A rollback restores code and config only — bound resources are not changed, so a D1 schema or KV data written by the newer version stays.

```
npx wrangler versions upload
npx wrangler versions deploy \
  NEW_ID@10% OLD_ID@90% -y
npx wrangler rollback  # the previous version
```

> **Gotcha:** Bindings and `vars` are not inherited by environments. A KV namespace declared at the top level does not exist in `env.staging` until you declare it there too — otherwise `env.CACHE` is `undefined` in staging.

Further reading

- Cloudflare Workers documentation
- Workers best practices
- Workers platform limits
- Wrangler configuration reference
- Rules of Durable Objects
- Queues: batching, retries, and delays