Cloudflare Workers Course

Flavio Copes

Updated August 7, 2026

Contents

Build and operate a Worker application with Hono, static assets, bindings, D1, KV, R2, Queues, scheduled work, Durable Objects, tests, logs, and safe deployments.

What you'll learn: Build, test, deploy, observe, and recover Link Vault, a full Worker application using the right Cloudflare capability for each job.

Workers foundations

Understand isolates, create the project, handle requests, and use Wrangler deliberately.

Understand the Workers runtime

Compare a Worker isolate with a long-running Node.js server or container and choose workloads that fit its request-driven model.

Cloudflare Workers run JavaScript and TypeScript in V8 isolates distributed across Cloudflare's network. They do not boot your own virtual machine or container for each request.

Workers use web-standard APIs such as Request, Response, URL, fetch, streams, and Web Crypto. Execution is event-driven and bounded, so keep request work focused and move durable state into bindings rather than process memory.

An isolate may handle many requests and may disappear at any time. Module-level constants and immutable lookup tables are fine, but an in-memory counter, cache, or login session is neither durable nor isolated to one user. Concurrent requests can also share an isolate, so global mutable state creates race conditions rather than coordination.

There is no durable local disk to treat as application storage. Use bindings for persistence and measure expensive work in terms of both CPU time and external waits. If a workflow must survive beyond one request, move it to a Queue, Durable Object, or another durable system instead of hoping the isolate stays alive.

List which parts of the Books API are portable web-standard code and which depend specifically on Node.js or local files.

Create the Link Vault project

Scaffold a TypeScript Worker with the current Cloudflare project generator and inspect the generated configuration and scripts.

The course project is Link Vault, a small service that saves useful links, serves a web interface, and exports backups.

Use Cloudflare's current project generator and select a Worker starter. The generator records current tool versions and a compatibility date, so prefer its output to hand-copying an old configuration example.

```
npm create cloudflare@latest -- link-vault
cd link-vault
npm run dev
```

The generated project is also a record of assumptions: package versions, module format, compatibility date, entry point, and local test setup. Commit those files together so another developer can reproduce the same baseline. Do not copy a compatibility date from an old tutorial; it controls runtime behavior and should be advanced intentionally with tests.

Before changing application code, run the generated test and a dry deployment. If either fails, fix the baseline first. Record the local URL, send one request with `curl`, and inspect the response headers so later routing or binding failures have a known-good comparison.

Open the local URL and identify `src/index.ts`, `wrangler.jsonc`, generated types, and test configuration.

Handle the first request

Read a standard Request and return JSON from the module Worker fetch handler using explicit status and headers.

A module Worker exports an object whose `fetch` method receives each incoming HTTP request, environment bindings, and an execution context.

Build responses with web APIs. Parse the URL once, route known paths, and return a real 404 for everything else. Avoid performing network or binding work at module initialization.

```
export default {
  async fetch(request: Request): Promise<Response> {
    const url = new URL(request.url)
    if (url.pathname === '/api/health') {
      return Response.json({ ok: true })
    }
    return Response.json({ error: 'Not found' }, { status: 404 })
  }
}
```

The handler's three inputs define the request boundary: `request` carries user-controlled HTTP data, `env` provides configured capabilities, and the execution context manages work that may outlive the response. Keep module initialization limited to deterministic setup; resource access there can run before the request and can be reused in ways you did not expect.

Return a response on every path and distinguish client errors from server errors. Parse a request body only on routes and methods that need it, enforce a size and shape, and attach a request ID to both the response and safe error logs. Test evidence should include status, content type, and body:

```
curl -i http://localhost:8787/api/health
curl -i http://localhost:8787/missing
```

Add the health route and inspect it with curl, including headers and an unknown path.

Use Wrangler development tools

Develop locally, regenerate binding types, validate configuration, and distinguish local resources from remote production state.

Wrangler connects source, configuration, local emulation, resource bindings, deployment, and logs.

Run local development by default and use remote resources only deliberately. Regenerate types after binding changes. Keep local and production data separate so a development command cannot mutate real users' state by surprise.

```
npx wrangler dev
npx wrangler types
npx wrangler deploy --dry-run
```

Every command has a target. Read the active Wrangler environment and binding output before running a mutation, especially when local and remote resources share similar names. Local emulation is the fast default; use a remote resource only for a test that depends on its real behavior and use disposable data.

A dry run validates packaging and configuration without publishing the Worker. It does not prove

that production secrets exist, remote migrations ran, or downstream services are reachable. Keep a short deployed smoke check for those boundaries. When diagnosing a deployed request, `wrangler tail` is useful live evidence, but it is not stored history and can be sampled, so production observability must not depend on an open terminal.

Change one configuration variable, regenerate types, and run a dry deployment validation.

Check your understanding: workers foundations

Check the isolate runtime, fetch handler, standard web APIs, Wrangler, local development, compatibility dates, and stateless execution.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Routing and configuration

Reuse Hono, configure typed bindings, separate secrets, and serve static assets.

Reuse Hono routes

Bring the web-standard routing and validation patterns from the Web APIs course into a Cloudflare Worker entry point.

Hono runs natively on Workers because its core uses Request and Response rather than Node-specific server primitives.

Create a Hono app, define `/api/links` routes, and export the app. Keep platform-independent validation and problem-response helpers in ordinary modules; use bindings only in repository adapters.

```
import { Hono } from 'hono'

const app = new Hono<{ Bindings: Env }>()
app.get('/api/health', c => c.json({ ok: true }))
export default app
```

Hono should own HTTP concerns: route matching, middleware, validation errors, and response formatting. Repository adapters should own D1 or KV operations. This boundary lets pure validation tests run without Cloudflare resources while runtime tests exercise the real binding adapters.

Middleware order is observable behavior. Put request IDs and error handling around routes, authenticate before protected handlers, and ensure the final 404 still returns the API's JSON error shape. Avoid catching every exception inside a route and returning 200; preserve meaningful status codes and log the internal error once with safe context.

Add collection and detail route placeholders with the same statuses and errors used in the Books API.

Configure bindings and types

Declare platform resources in wrangler.jsonc and access them through a typed environment instead of global variables or credentials.

A binding gives Worker code direct access to a Cloudflare resource such as D1, KV, R2, a Queue, or another service.

The binding name becomes a property on `env`. Configuration defines which resource sits behind that name in each environment. Run `wrangler types` after changes and avoid hand-maintaining a second type definition that can drift.

```
{
  "d1_databases": [
    { "binding": "DB", "database_name": "link-vault", "database_id": "..." }
  ]
}
```

A binding is capability injection: code receives access to one named resource without handling that resource's credentials. Keep the binding name stable across environments while pointing it at separate development, staging, and production resources. Then application code stays the same and configuration expresses the deployment boundary.

Generated types prove that code and configuration agree at compile time, not that the resource exists or contains the expected schema. Add a startup or smoke request that exercises each critical binding. Also avoid creating a secret-derived client once in global scope. A binding-only deployment may reuse an isolate, leaving that object configured with an older secret; create it from the current `env` inside the request instead.

Declare the D1 binding created in the next module and generate the environment types.

Separate variables and secrets

Put ordinary environment configuration in versioned vars and sensitive values in the Cloudflare secret store.

Not every setting is a secret. Public environment names and feature choices can live in configuration; credentials must not.

Access both through `env`, but create secrets with Wrangler or the dashboard so plaintext never enters Git. Configure each environment separately because bindings and secret values may not inherit the way you expect.

```
npx wrangler secret put API_TOKEN
npx wrangler secret list
```

Classify by impact, not by convenience. A feature flag or public environment name is a variable; an API token, signing key, or private endpoint credential is a secret. Both arrive through `env`, but only ordinary configuration belongs in version control.

Use separate credentials with the smallest permissions for each environment. Rotation means adding the new credential, deploying code that reads it, verifying the dependent path, and revoking the old credential. A deployment rollback can reactivate older code, so check that the rollback version does not require a revoked secret. Logs, exceptions, response bodies, and local environment files are all possible leak paths even when Git is clean.

Add a non-secret `ENVIRONMENT` variable and a practice secret, then confirm code can read them without printing the secret.

Serve static assets

Attach the Link Vault browser interface as Worker static assets while keeping API routes and asset fallback behavior explicit.

A Worker project can deploy static assets with the application so one release contains both the interface and API.

Configure an assets directory and decide whether asset handling happens before or after Worker routing. Keep `/api/*` responses out of an HTML fallback, and use content-hashed assets or deliberate caching for updates.

Write the routing contract down before deploying it. A missing JavaScript file should be a 404, not the application's HTML shell. An unknown browser route may use an HTML fallback, while an unknown API route must keep its JSON status and content type. Test all three cases directly rather than checking only the homepage.

Treat HTML and fingerprinted assets differently. HTML should update quickly because it points to the current filenames; content-hashed JavaScript and CSS can be cached much longer because a content change creates a new URL. A rollback must restore a compatible HTML-and-assets pair, which is easiest when both ship in the same versioned deployment.

Create an accessible link form and list page, bind the assets directory, and verify `/` and `/api/health` return different content types.

Check your understanding: routing and configuration

Check Hono routing, bindings, variables, secrets, static assets, environments, and Worker-specific configuration boundaries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Storage bindings

Store relational data in D1, cache with KV, export to R2, and choose storage well.

Store links in D1

Create a D1 database and migration, then use prepared statements and bound values for Link Vault CRUD operations.

D1 provides managed SQLite semantics through a Worker binding. It fits Link Vault's relational link records and queries.

Create the database, generate a migration, apply it locally, and use `prepare().bind()` for every request value. Keep local and remote migration commands explicit; a migration applied locally has not changed production.

```
npx wrangler d1 create link-vault
npx wrangler d1 migrations create link-vault create-links
npx wrangler d1 migrations apply link-vault --local
```

Make constraints carry business invariants that should survive every code path: primary keys,

required fields, and any uniqueness rule belong in the schema. Prepared statements with bound values separate SQL from user input, but they do not replace URL validation or authorization.

Keep migration state explicit. Apply migrations to an isolated local database in tests, then use the remote command only after confirming the account, environment, and database name. Prefer additive changes during a deployment window so old and new Worker versions can both operate. If a change cannot be reversed, define a data recovery step before applying it rather than calling a code rollback the whole plan.

Create links with ID, URL, title, created timestamp, and archived flag, then implement list and create routes.

Cache read-heavy data in KV

Use KV for read-heavy derived values while respecting eventual consistency and keeping D1 as the authoritative record.

Workers KV distributes values for fast reads, but changes may take time to become visible everywhere.

That consistency model fits cache entries and configuration better than rapidly updated counters or unique constraints. Cache the public recent-links response with an expiration and invalidate it after writes, while accepting that a reader may briefly see the previous value.

```
const cached = await env.CACHE.get('recent-links', 'json')
if (cached) return Response.json(cached)
const links = await listRecentLinks(env.DB)
await env.CACHE.put('recent-links', JSON.stringify(links), { expirationTtl: 60 })
```

KV can return a previously cached value after a write, including a cached ”not found.” Deleting or overwriting a key is therefore best-effort invalidation, not immediate global coordination. Keep D1 authoritative and make correctness independent of seeing the newest KV value.

Version cache keys when the response shape changes, for example `recent-links:v2`, so old data cannot be decoded as a new format. Choose the TTL from the acceptable staleness window, not from a desire for a high hit rate. If many misses can arrive together, cap the database work or add coordination; a cache miss storm can otherwise move the load rather than remove it.

Add the cache and log safe hit-or-miss metadata during local testing.

Store exports in R2

Generate a JSON backup and store it as an R2 object instead of placing large binary or file content in D1 or KV.

R2 is object storage for files and large values. Link Vault will write versioned export objects and stream them back on request.

Use a structured object key, set content-type metadata, and stream object bodies instead of buffering unnecessary copies. Store relational metadata in D1 only if the application needs to query exports independently of their object keys.

```
const key = `exports/${userId}/${crypto.randomUUID()}.json`
await env.EXPORTS.put(key, JSON.stringify({ links }), {
  httpMetadata: { contentType: 'application/json' }
```

```
})
```

An object key is an address, not an authorization rule. Resolve the current user first, then derive or look up only keys that user may access. Do not let an arbitrary path parameter become unrestricted bucket access. Preserve content type and a download filename, and stream `object.body` so the Worker does not create another full in-memory copy.

Write the object before marking an export ready in D1, or use an intermediate state such as `pending`. If either operation fails, the state should reveal what can be retried or cleaned up. Retention and deletion are also application behavior: define whether replacing a database record deletes the old object, and test an unknown, unauthorized, and successfully streamed export separately.

Write one export, read it through a protected route, and return 404 for an unknown key.

Choose the right storage product

Match relational queries, read-heavy keys, objects, and strongly coordinated state to D1, KV, R2, or Durable Objects.

Cloudflare storage products overlap at the level of "save data," but differ in consistency, query model, size, and coordination.

Use D1 for relational records, KV for high-read values tolerant of stale propagation, R2 for object bodies, and Durable Objects for strongly coordinated per-entity state. Recheck current product docs, limits, and pricing before committing a production design.

Start with the invariant, then choose the product. "A user can never claim the same name twice" needs coordinated or transactional enforcement. "A public list may be one minute old" can use an eventually consistent cache. "Download this exact export" fits an object key. Querying by several fields points toward relational storage.

Queues are delivery, not the source of truth, and secrets belong in bindings rather than any of these stores. For every datum, name the authoritative copy, acceptable staleness, expected access pattern, size, retention, and recovery method. If two products hold copies, define which one wins and how the derived copy is rebuilt.

Classify link records, cached public lists, export files, live collaboration rooms, secrets, and queue messages.

Check your understanding: storage bindings

Check D1 prepared statements and migrations, KV consistency, R2 object storage, product selection, and data ownership across bindings.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

State and background work

Use waitUntil, Queues, Cron Triggers, and Durable Objects for the jobs they fit.

Defer small work with waitUntil

Keep a request response fast while allowing a short non-critical promise to continue through the execution context.

Some work should not delay the response, such as writing non-critical analytics or invalidating a cache after a successful authoritative write.

`ctx.waitUntil()` extends the lifetime of the request event for a promise. It is not a durable job queue and does not turn long or critical workflows into reliable background processing. Await work required for a correct response.

Pass the promise itself to `waitUntil`; merely starting an async function leaves its lifecycle untracked. Handle and log rejection with a request ID, because the client has already received a response and cannot be told that the follow-up failed.

The decision test is simple: if losing the work would make accepted data incorrect, create financial or user-visible inconsistency, or require retries, use a durable workflow such as a Queue. Cache invalidation and optional analytics can tolerate best-effort execution. Store the D1 row first, schedule invalidation second, and make stale cache data harmless if invalidation never completes.

Invalidate the recent-links cache with waitUntil only after D1 successfully creates a link.

Process exports with Queues

Move export generation to an at-least-once queue consumer with explicit acknowledgements, retries, and idempotent output.

Generating a large export need not hold an HTTP request open. The API can enqueue a job and return 202 with a job identifier.

Queue delivery is at least once, so a message may be processed again. Derive an idempotent R2 key from the job ID, record job state, and handle each message error individually so one failure does not repeat unrelated successful work.

Treat the message as a versioned contract. Include a job ID, user ID, requested format, and schema version—not a large copy of data that can become stale. The producer should persist a pending job before enqueueing and return `202 Accepted` with a status URL. The consumer moves that job through explicit states.

Idempotency means a retry converges on the same result. Check whether the deterministic object already exists or safely overwrite it, then mark the job complete. Separate permanent validation failures from temporary dependency failures, configure retry and dead-letter handling, and log the message and job IDs without logging private export contents.

Create an export job, send it to a bound queue, and have the consumer write one predictable R2 object.

Run scheduled maintenance

Add a scheduled handler for bounded cleanup work and test it without relying on an incoming HTTP request.

Cron Triggers invoke a Worker's scheduled handler according to configured UTC schedules.

Use schedules for periodic tasks such as deleting expired export metadata in bounded batches. Keep

the job idempotent, log a safe summary, and store progress if it cannot finish within one invocation.

```
export default {
  fetch: app.fetch,
  async scheduled(event: ScheduledEvent, env: Env, ctx: ExecutionContext) {
    ctx.waitUntil(deleteExpiredExports(env))
  }
}
```

Schedules are expressed in UTC and are delivery triggers, not precise wall clocks. A run can be delayed or overlap a previous run, so cleanup must be safe to repeat. Use a stable cutoff passed into the operation and delete a bounded page of rows rather than scanning the entire dataset in one invocation.

If more work remains, store a cursor or enqueue the next bounded unit. Include the scheduled timestamp, cursor, deleted count, and error class in logs. Test the handler directly with a fixed clock and fixtures on both sides of the cutoff; waiting for the next real cron event is not a useful development loop.

Add a local scheduled test that removes expired rows but preserves active exports.

Coordinate with Durable Objects

Recognize when one globally unique stateful object is the right tool for per-user coordination, locks, counters, or real-time rooms.

A Durable Object combines an addressable unique instance with strongly consistent storage and serialized access for that object.

Use a deterministic ID when all requests for one user or room must meet the same object. Persist critical state because the object can hibernate or restart. Do not route all global traffic through one object when the workload can be sharded.

The ID defines the coordination boundary. A per-user ID serializes one user's limiter without making unrelated users wait; a single global ID would create a bottleneck and a larger failure domain. Authenticate before selecting the object and derive the ID from a stable internal identifier, not an attacker-controlled label.

Serialized access does not make in-memory fields durable. Save the decision state in Durable Object storage before returning success, and design alarms or retries to be idempotent. Also remember that calls to external systems can fail after local state changes. Record enough state to resume or compensate instead of holding an implicit lock across unreliable network work.

Design a per-user export limiter that coordinates concurrent export requests without relying on eventually consistent cache data.

Check your understanding: state and background work

Check waitUntil, Queues delivery and idempotency, scheduled handlers, Durable Object coordination, persistence, and retries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Production and deployment

Test the runtime, add observability, deploy compatibly, roll back, and review production.

Test with the Workers runtime

Run integration tests inside the Workers execution environment with local bindings instead of mocking every platform API.

Ordinary unit tests still suit pure Link Vault logic. Runtime integration tests should exercise the actual Worker module and local D1, KV, R2, and Queue behavior.

Use Cloudflare's Vitest integration and isolated storage for tests. Apply migrations, send requests through the Worker, and assert binding effects. Keep a small remote verification step for differences that local emulation cannot represent.

Split tests by evidence. Pure tests prove validation and state transitions. Runtime integration tests prove that the exported handler, generated environment, migrations, and binding APIs work together. A deployed smoke test proves that the published version is connected to the intended remote resources.

Reset storage between tests and never let test order create the fixture. For eventual-consistency behavior, test the application's tolerance rather than expecting local emulation to reproduce global propagation timing. Exercise failure paths too: a missing migration, duplicate queue delivery, absent R2 object, and unavailable downstream service should produce controlled states and useful logs.

Test create/list in D1, a KV cache miss and hit, an R2 export, and duplicate queue processing.

Add observability

Use structured logs, traces, request IDs, and platform metrics without leaking secrets, authorization values, or personal link data.

Distributed code needs enough context to connect a user-visible failure to one invocation and its downstream operations.

Enable observability deliberately, log structured safe fields, and propagate a request ID. Record route templates, status, duration, binding operation type, and error class—not raw tokens, secrets, or full private URLs.

Logs answer what happened in one execution; metrics reveal rate and trend; traces connect time spent across operations. Use all three for different questions. Include the Worker version or deployment ID so a spike can be tied to a release, and use the same job ID across the HTTP producer and Queue consumer.

`wrangler tail` is valuable for a live investigation, but events can be sampled and the stream is not durable storage. Configure retained Workers Logs and alerts for production signals. Deliberately trigger one safe 404 and one controlled dependency error, then confirm an operator can find them without searching for private user content.

Add one request log and one queue-job log, then inspect local output and a deployed tail.

Deploy, migrate, and roll back

Ship a versioned Worker and its data changes in a compatible order with a tested recovery path.

Code rollback is straightforward only when the database and message formats remain compatible with both versions.

Use additive D1 migrations before code that depends on them, deploy with a clear version message, run smoke checks, and keep the previous version available. Make queue consumers tolerate messages created by the previous producer during rollout.

Record the published version ID and the exact checks that passed. A useful sequence is expand the schema, deploy tolerant code, observe it, and remove old fields only in a later release after the rollback window. Version Queue messages so an older consumer can reject or safely ignore a format it cannot process.

`wrangler rollback` changes the active Worker deployment, but it does not roll back D1 data, KV values, R2 objects, Queue effects, or external APIs. It may also be unavailable when required bindings or Durable Object classes no longer match the old version. Test the rollback while those dependencies still exist, then smoke-test the restored version and reconcile Git so the next deployment does not reintroduce the failure.

Write a release sequence for adding an optional link description, including migration, code, smoke test, and rollback.

Complete the production review

Review compatibility dates, runtime limits, costs, permissions, storage semantics, recovery, and ownership before calling Link Vault production-ready.

Platform capabilities and plan limits evolve. The production review must consult current official documentation rather than trust numbers copied into an old checklist.

Confirm the compatibility date, binding targets, environment separation, secret access, resource limits, current pricing, cache consistency, D1 recovery, queue retries, R2 lifecycle, logs, alerts, and a tested rollback. Remove unused products and permissions.

Turn the checklist into evidence: the deployed version ID, binding names and resource owners, latest restore test, smoke-test output, alert destination, and the person who can execute recovery. A checkbox without a command, link, or observation quickly becomes ceremonial.

Review failure boundaries as a whole. Code rollback does not restore data, cache invalidation is not immediate, background work can be delivered more than once, and an expired credential can break an older version. Set budgets and alerts from current plan documentation, then remove resources, routes, secrets, and permissions the application no longer needs. Finish by having someone other than the author follow the runbook.

Run the checklist against staging, deploy a version, create and export one link, inspect logs, then exercise the rollback plan.

Check your understanding: production and deployment

Check Worker integration tests, observability, safe logging, deployment versions, migrations, rollback, limits, compatibility, and production review.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/cloudflare-workers/.