Web APIs Course

Flavio Copes

Updated August 3, 2026

Contents

Design and build a production-shaped TypeScript API with Hono, SQLite, validation, OpenAPI, consistent errors, security boundaries, tests, and operational logs.

What you'll learn: Build, document, test, secure, and prepare a resource-oriented Books API for a compatible production release.

API foundations

Design the contract, start Hono on Node.js, return JSON, and inspect HTTP exchanges.

Design the Books API

Turn a small product requirement into resources, representations, routes, and a stable HTTP contract before writing handlers.

An API is an interface between programs. Our project is a Books API that lets clients list, create, read, update, and delete books.

Model nouns as resources and use HTTP methods to express common actions. Start with `/books` and `/books/:id`. Write one example JSON representation and the expected response status for each operation before choosing database code.

The contract is more than route names. Decide which fields the client may send, which fields the server owns, whether an update replaces or partially changes a book, and what an empty collection looks like. Keep one stable representation shape so clients do not need route-specific guesses.

Write a small behavior table before implementation: method and path, input, successful status, response body, and expected errors. Include `GET /books`, `POST /books`, `GET /books/:id`, `PUT /books/:id`, and `DELETE /books/:id`. This table becomes the basis for tests and OpenAPI later. If a handler decision cannot be traced to the table, the contract is still incomplete.

Sketch the five routes, one book representation, and the error response for an unknown book.

Create a Hono Node project

Scaffold a TypeScript API on Node.js with Hono and run the local development server.

Hono builds on the web-standard Request and Response APIs and can run on Node.js or edge runtimes with the appropriate adapter.

Use the official project generator and select the Node.js template. The generated dependency versions become the project's source of truth, so read its package.json rather than copying a version from the lesson.

```
npm create hono@latest books-api
# Select the nodejs template
cd books-api
npm install
npm run dev
```

Keep the Hono application separate from the Node.js adapter. The app owns routes and returns standard `Response` objects; the adapter owns the TCP server, port, and shutdown lifecycle. That boundary lets tests call `app.request()` without opening a socket and makes most application code portable to another Web API runtime.

Inspect the generated scripts, lockfile, Node requirement, and entry point before changing them. Start from a clean install, request the local URL with `curl -i`, and note the listening port. If the command starts but the request fails, distinguish a compile error, a server bound to a different port, and a handler returning an error instead of reinstalling dependencies blindly.

Start the server and find the exported Hono app, Node adapter entry point, and TypeScript configuration.

Return the first resource

Create a GET route that returns a predictable JSON collection and an explicit HTTP success response.

Begin with an in-memory collection so the first route tests only the API layer.

Hono's context creates a JSON Response with the correct content type. Keep response data consistent: return a named collection rather than sometimes returning an array and sometimes an object.

```
import { Hono } from 'hono'

const app = new Hono()
const books = [{ id: '1', title: 'Dune', author: 'Frank Herbert' }]

app.get('/books', c => c.json({ books }))

export default app
```

The envelope `{ books: [...] }` leaves room for pagination links or metadata without changing the top-level type later. An empty collection is still a successful collection response: return `200` with `{ books: [] }`, not `404` and not a different shape.

Treat the response as observable evidence. Confirm the status, `content-type`, and exact JSON keys. Mutation of the in-memory array will be shared by later requests in this one server process, which is useful for learning but not durable storage. Restart the process and verify that the seed data returns; that failure is the reason the data module later moves behind a database boundary.

Add the route and request it with curl and the browser network panel.

Inspect requests and responses

Use curl to see methods, URLs, headers, bodies, and status codes instead of debugging an API from rendered JSON alone.

A browser tab hides most of an HTTP exchange. API debugging starts by inspecting the complete request and response.

`curl -i` includes response headers. `-v` shows connection and request details. Use `-X` only when necessary; a request body option such as `--json` already selects POST.

```
curl -i http://localhost:3000/books
curl -i --json '{"title":"Dune","author":"Frank Herbert"}' http://localhost:3000/books
```

Read the exchange in order: resolved URL, request method and headers, response status, response headers, then body. A JSON-looking body does not prove success; a proxy can return an HTML error, a redirect, or a cached response with a surprising status. Use `-i` for contract evidence and add `-v` when connection, TLS, redirect, or request-header details matter.

The browser network panel adds browser-specific evidence such as preflight requests and blocked response access. The server may have processed a request even when browser JavaScript reports a CORS failure. Reproduce the HTTP request with `curl`, then compare the browser's `Origin` and preflight headers before changing application logic.

Send one valid request and one request to a missing path. Save the exact status, content type, and

body for each.

Check your understanding: api foundations

Check resources, representations, contracts, Hono setup, JSON responses, and request inspection.

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

Take this interactive quiz online

Routes and errors

Read parameters, create and change resources, and return consistent problem responses.

Read path and query parameters

Use path parameters to identify one resource and query parameters to filter or shape a collection.

Path and query parameters solve different problems. `/books/42` identifies one book; `/books?author=le-guin` asks for a filtered collection.

Read path values with `c.req.param()` and query values with `c.req.query()`. Both arrive as strings and both are untrusted input. Return 404 when a valid identifier names no resource.

```
app.get('/books/:id', c => {
  const id = c.req.param('id')
  const book = books.find(item => item.id === id)
  if (!book) return c.json({ type: 'about:blank', title: 'Book not found', status: 404 }, 404
  return c.json({ book })
})
```

Separate malformed input from an absent resource. If IDs have a required shape, reject a malformed value with `400`; return `404` only after a valid ID has been looked up and is missing. A filter that matches no books is different again: it returns `200` with an empty collection.

Define normalization explicitly. Decide whether author matching is exact or case-insensitive, whether surrounding whitespace is meaningful, and what repeated or empty query parameters do. Do not copy a query value into SQL or a response header before validation. Test encoded characters, an empty author, a malformed ID, and a valid missing ID so routing, decoding, validation, and lookup failures remain distinguishable.

Add the detail route and an optional author filter to the collection route.

Create resources with POST

Parse a JSON request, create a server-owned identifier, and return 201 with the new resource and Location header.

POST to a collection asks the server to create a subordinate resource. The server decides the new resource URL.

Parse JSON only when the content type is correct, create the identifier on the server, and return `201 Created`. A `Location` header lets the client find the canonical new resource.

```
app.post('/books', async c => {
```

```
  const input = await c.req.json()
  const book = { ...input, id: crypto.randomUUID() }
  books.push(book)
  c.header('Location', `/books/${book.id}`)
  return c.json({ book }, 201)
})
```

Do not spread unknown input into a stored object in production code. Validate and pick the writable fields first, then assign server-owned fields such as `id` and `createdAt` afterward so a client cannot override them. Persist the resource before returning `201`; a successful response is a promise that the canonical URL can be read.

Classify failures at the boundary. An unsupported content type can return `415`, malformed JSON is a `400`, and syntactically valid JSON that violates the book schema can return `422`. A retry after an uncertain network failure can create a duplicate, so clients must not assume POST is idempotent. The later idempotency lesson adds a deliberate retry contract.

Create a book, follow the Location header, and confirm a subsequent list includes it.

Update and delete resources

Implement predictable update and deletion behavior without treating every action as an unrelated POST endpoint.

Use the standard method semantics when they fit. `PUT /books/:id` replaces the client-editable representation; `DELETE` removes it.

Choose and document whether PUT can create a missing resource. For this project it cannot, so a missing ID returns 404. A successful DELETE can return 204 with no response body.

```
app.delete('/books/:id', c => {
  const index = books.findIndex(book => book.id === c.req.param('id'))
  if (index === -1) return c.json({ title: 'Book not found', status: 404 }, 404)
  books.splice(index, 1)
  return c.body(null, 204)
})
```

`PUT` represents a full replacement of the client-editable state. Missing writable fields should therefore fail validation rather than silently retain their previous values. If the API later needs partial changes, add `PATCH` with a separate contract instead of making `PUT` mean both things.

Keep server-owned fields such as ID and creation time unchanged, and perform lookup, authorization, and mutation as one coherent operation when the database arrives. A `204` response has no body; do not attach JSON or a content type to it. Repeat the same PUT and DELETE during the exercise: replacement with the same representation should converge on the same state, while a second delete should produce the documented missing-resource response.

Add PUT and DELETE, then repeat each request to see whether its semantics remain understandable.

Standardize API errors

Return one useful problem-details shape for validation, missing resources, conflicts, and unexpected failures.

Clients should not need a different parser for every error. Standard fields make failures machine-readable and still understandable to a person.

Use the problem-details fields `type`, `title`, `status`, `detail`, and optional application-specific extensions. Do not expose stack traces, SQL, secrets, or internal file paths. Log unexpected detail on the server with a request identifier.

```
{
  "type": "https://example.com/problems/invalid-book",
  "title": "Book input is invalid",
  "status": 422,
  "errors": { "title": "Required" }
}
```

Serve problem details with `application/problem+json` and make the HTTP status match the `status` member. `title` describes the problem class and should remain stable; `detail` can explain this occurrence without exposing internals. A stable `type` URI lets clients recognize a class without parsing English text.

Use extensions for structured data such as field errors and a request ID. Clients must still tolerate new extension fields. Centralize unexpected exceptions so they become one generic `500` response and one detailed server log, not several leaked stack traces. Test the helper's status, media type, required fields, and absence of SQL, file paths, and secret values.

Create one helper that returns problem JSON and use it for 404 and invalid-input responses.

Check your understanding: routes and errors

Check path and query parameters, creation responses, replacement and deletion semantics, status codes, and consistent error documents.

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

Take this interactive quiz online

Validation and data

Validate runtime input, persist SQLite data, bind SQL values, and paginate collections.

Validate unknown input

Treat request data as unknown at runtime and turn invalid JSON fields into precise client-facing validation errors.

TypeScript cannot prove what a remote client sent. A type assertion changes the checker's belief but performs no runtime validation.

Define a schema for writable book fields and validate before business logic. Reject missing, incorrectly typed, too-long, or unexpected values according to one documented policy. Hono validation middleware can attach the validated result to the request context.

Validation begins before the schema. Limit body size, require the expected media type, and turn JSON parse failures into a controlled client error. Then validate the parsed value as `unknown`. TypeScript types describe code after that boundary; they do not make a network payload trustworthy.

Use only the validated output in the handler. Decide whether unknown fields are rejected or stripped, and keep that policy consistent so misspelled input cannot appear to succeed. Normalize only where the contract says so: trimming a title may be useful, while silently coercing `"2024"` into a number hides a client bug. Test null, arrays, extra fields, whitespace-only strings, invalid years, and the largest accepted input.

Add runtime validation for non-empty title and author strings and an optional four-digit publication year.

Design the SQLite schema

Move books from memory into a constrained SQLite table whose schema preserves the API's important invariants.

In-memory arrays disappear on restart and cannot coordinate several server processes. SQLite gives the local API durable transactional storage.

Use a text primary key, required title and author fields, and timestamps stored in one consistent format. Database constraints are a second line of defense; they protect writes made outside the HTTP handler too.

```
CREATE TABLE books (
  id TEXT PRIMARY KEY,
  title TEXT NOT NULL CHECK (length(title) > 0),
  author TEXT NOT NULL CHECK (length(author) > 0),
  published_year INTEGER,
  created_at TEXT NOT NULL
);
```

The schema is the last local authority for data integrity. HTTP validation gives clients friendly errors; constraints prevent a background script, test helper, or future route from inserting impossible rows. Keep timestamps in one UTC representation and generate server-owned values in one place.

Treat schema changes as versioned migrations, not startup-time guesses. Apply the same migration files in development, tests, and deployment, and record which ones completed. For a write that spans several statements, use a transaction so either the entire state change commits or none of it does. SQLite allows concurrent readers but coordinates writes, so keep transactions short and handle a busy or constraint failure without returning raw database text.

Create the database and table, then insert one valid row and one row that violates a constraint.

Write parameterized queries

Keep SQL structure separate from untrusted values and translate database outcomes into API responses.

Never build SQL by concatenating request values. Quoting by hand is not a reliable security boundary.

Write the SQL structure with placeholders and bind the values through the database library. Handle expected constraint errors explicitly, and let unexpected errors reach centralized logging rather than returning raw database messages.

```
INSERT INTO books (id, title, author, published_year, created_at)
VALUES (?, ?, ?, ?, ?)
```

Parameters protect value positions. They cannot safely stand in for SQL identifiers or keywords such as a column name or sort direction. If a client can choose sorting, map a small allowlist of API values to fixed SQL fragments rather than interpolating the raw query string.

Translate expected database outcomes at the repository boundary. A uniqueness or check violation can become a documented conflict or validation problem; a missing row becomes `404`; an unexpected I/O or programming error remains an internal failure. Log the error class and request ID, not the full SQL plus bound personal values. Test a title containing quotes and SQL-looking text—the value should be stored literally and the table must remain intact.

Replace the in-memory create and detail operations with prepared parameterized statements.

Paginate, filter, and sort

Keep collection responses bounded and stable with validated limits, cursors or offsets, filters, and deterministic ordering.

An unbounded collection route becomes slower and more expensive as data grows. Pagination is part of the contract from the start.

For the first version, accept a capped `limit` and non-negative `offset`, filter by author, and order by creation time plus ID for a deterministic tie-breaker. Return pagination metadata or links so a client knows how to continue.

```
curl "http://localhost:3000/books?author=Le%20Guin&limit=20&offset=0"
```

Validate before constructing SQL: reject non-integers and negative offsets, apply a modest default, and cap the maximum limit. Use bound parameters for filter values and an allowlist for sort fields. Add indexes that match real filters and ordering only after inspecting the query and dataset; every index also adds write and storage cost.

Offset pagination is simple but rows inserted or deleted between requests can shift later pages. Deterministic ordering limits ambiguity but does not freeze the dataset. For a busy collection, move to a cursor containing the final sort values, such as `created_at` and `id`. Whichever strategy you publish, treat it as contract data: return the next link or cursor, and return an empty successful page when the client reaches the end.

Add validation and parameterized SQL for the collection query, then request two small pages.

Check your understanding: validation and data

Check runtime validation, SQLite schema design, parameterized queries, constraints, pagination, filtering, and deterministic ordering.

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

Take this interactive quiz online

Contracts and security

Describe OpenAPI, configure CORS, place auth boundaries, and handle retries and abuse.

Describe the API with OpenAPI

Write a machine-readable contract for routes, parameters, request bodies, responses, and reusable schemas.

OpenAPI describes an HTTP API independently of one client or server implementation. It supports documentation, validation, client generation, and review.

Start with the paths and schemas the API already implements. Name every response status and media type. Keep the document in version control and test it against behavior so it does not become an attractive but false contract.

Describe the wire format, not TypeScript implementation types. Mark required properties, distinguish nullable values from missing properties, specify parameter locations, and reuse schemas without hiding meaningful differences between create input and a stored Book. Examples help readers but do not constrain a value unless the schema does.

Pin the OpenAPI version declared by the document and validate it with a current parser in CI. Contract tests should exercise representative success and known error responses, including headers such as `Location` and the problem-details media type. Review generated clients as consumers: a surprising optional field or undocumented status is evidence that the description and server have drifted.

Describe GET and POST `/books`, including the Book schema, create input, 200, 201, 422, and problem responses.

Configure CORS deliberately

Allow browser clients from known origins without confusing CORS with authentication or general server security.

Browsers restrict scripts from reading cross-origin responses unless the server grants permission through CORS response headers.

Allow only the methods, headers, origins, and credential behavior the browser application needs. CORS does not stop curl or another server from making a request, so it is never a replacement for authentication or authorization.

An origin is the scheme, host, and port tuple, not a path or loose hostname suffix. For a preflight, the browser sends `OPTIONS` with the intended method and headers; the API answers whether the real request may proceed. The preflight carries no credentials, even when the eventual request will.

If credentialed browser requests are allowed, return the exact approved origin rather than `*`, enable credentials deliberately, and vary the response by `Origin` so caches do not reuse permission for another site. Test an allowed origin, a rejected origin, and this explicit preflight:

```
curl -i -X OPTIONS http://localhost:3000/books \
  -H 'Origin: http://localhost:5173' \
  -H 'Access-Control-Request-Method: POST' \
  -H 'Access-Control-Request-Headers: content-type'
```

Add a development origin allowlist and inspect both a simple request and an OPTIONS preflight.

Place the authentication boundary

Separate identifying a caller from deciding whether that caller may perform one specific operation.

Authentication answers who is calling. Authorization answers whether that identity may access this book or operation.

Put authentication in middleware that produces a trusted identity, then enforce ownership or roles close to the resource query. Never accept a user ID from the request body as proof of identity. The later Web Authentication course builds the complete mechanism.

Middleware should either reject the request or attach a small trusted principal such as an internal user ID and roles. Handlers consume that principal; they do not reparse credentials or trust ownership fields from JSON. Keep the raw token out of logs and application objects.

Authorization belongs in the data operation. Query or update the book with both its ID and owner ID so another request cannot change ownership between a separate check and write. Use `401` when valid authentication is required and absent or invalid, `403` when an identified caller is forbidden, and optionally `404` when revealing that another user's resource exists would leak information. Test two users against the same book, not only the owner path.

Add a temporary test identity middleware and restrict updates to books whose owner ID matches it.

Handle retries and abuse

Use request limits, idempotency, and safe timeouts to make the API predictable under duplicate traffic and deliberate pressure.

Networks retry and users double-click. Attackers also send requests faster than a normal client. The API needs explicit policies for both cases.

Cap body size and request duration, rate-limit by an appropriate identity, and return useful 429 responses. For important POST operations, an idempotency key can let the server return the original result instead of creating a duplicate.

Apply cheap limits early. Reject an oversized body before parsing it, stop work when a timeout or client cancellation is signaled, and bound database and upstream calls too. An IP address may represent many users or be supplied through an untrusted proxy header, so choose the rate-limit identity from authenticated users or verified network metadata.

Store an idempotency key with the caller identity, a fingerprint of the relevant request, processing state, and the final status, headers, and body. A unique constraint makes simultaneous duplicates converge. Reusing the key with different input must fail rather than return an unrelated result. Define retention and failure recovery so a crashed in-progress record does not block the caller forever, then test sequential and concurrent duplicates.

Design an idempotency record for book creation and a small per-identity development rate limit.

Check your understanding: contracts and security

Check OpenAPI contracts, CORS, authentication and authorization boundaries, rate limits, idempotency, and safe API defaults.

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

Take this interactive quiz online

Test and ship

Test handlers and data, add safe observability, and prepare a compatible release.

Test handlers with requests

Exercise the Hono application directly with standard Request objects before adding network and database integration tests.

A Hono app exposes a fetch-compatible interface, so route tests can send requests without opening a TCP port.

Test observable status, headers, and body. Use a fresh application or injected repository per test so state cannot leak between cases. Avoid asserting private function calls when the HTTP result expresses the behavior better.

```
import test from 'node:test'
import assert from 'node:assert/strict'
import app from './app.js'

test('lists books', async () => {
  const response = await app.request('/books')
  assert.equal(response.status, 200)
  assert.deepEqual(await response.json(), { books: [] })
})
```

These tests cover routing, middleware, validation, and serialization without the timing and port noise of a live server. They do not prove the Node adapter or real database works, which is why later tests add those boundaries. Keep each layer's claim explicit.

Build a fresh app with an injected repository for every test. Assert externally visible behavior: status, media type, headers such as `Location`, and parsed body. Read a response body only once, just as a real `Response` stream requires. Add table-driven invalid inputs and verify that unexpected repository failures become a safe problem response rather than an unhandled rejection or leaked error.

Test list, create, invalid input, missing resource, and delete behavior with isolated state.

Add database integration tests

Run the real schema and queries against an isolated SQLite database to catch mistakes mocks cannot reveal.

A mock can prove that code made an expected call. It cannot prove that the SQL, constraint, migration, and driver agree.

Create a temporary database for each test file or case, apply migrations, call the API, and remove the file afterward. Cover rollback and constraint behavior as well as the happy path.

Use the same migration runner and repository code as production. Creating a simplified test schema by hand can let a broken migration ship. Give every test an isolated file or database connection, close statements and connections before cleanup, and never point the test configuration at a development database.

Assert persisted state after the HTTP response, not only the response body. Exercise a constraint

violation, a multi-step transaction that rolls back, a migration from an empty database, and any relevant concurrent write behavior. When a test fails, preserve enough context to distinguish an HTTP mapping bug from SQL, schema, locking, or cleanup failure.

Run create, read, update, and delete through the app against a temporary database.

Add request IDs and structured logs

Make one failing request traceable without logging secrets, passwords, authorization values, or complete personal data.

Useful logs answer which request failed, where, how long it took, and what the server decided.

Accept a valid incoming request ID or generate one, return it in a response header, and include it in structured logs. Record method, route, status, duration, and safe error classification. Redact sensitive headers and bodies.

Do not trust an arbitrary incoming identifier. Enforce a short length and safe character set, or replace it, so a caller cannot inject multiline log content or huge values. Log the route template such as `/books/:id` rather than a raw URL that may contain personal query data.

Place timing middleware around the whole request and emit the completion record even when a handler throws. Log an unexpected error once with its request ID and internal stack, then return a generic problem response carrying the same ID. Send one controlled failure and prove you can join the response header, completion log, and error log without searching for a title, token, or request body.

Add middleware that logs one JSON object after each request and one safe object for unexpected errors.

Prepare a compatible release

Configure, shut down, version, document, and review the API as a product contract rather than just a running server.

A deployable API reads configuration from its environment, reports health, stops gracefully, and changes its contract deliberately.

Validate required configuration at startup. On shutdown, stop accepting new traffic and finish or time out active work. Prefer additive contract changes; when a breaking change is unavoidable, publish a clear migration and version boundary.

Separate liveness from readiness. Liveness answers whether the process can run; readiness answers whether this instance should receive traffic after configuration and required dependencies are available. Keep both cheap and avoid making health probes mutate data. During `SIGTERM`, fail readiness first, stop new connections, allow bounded active work to finish, close the database, then exit with evidence if the deadline is exceeded.

A code rollback does not undo schema changes or external effects. Use expand-and-contract migrations so old and new versions overlap safely, and keep the previous artifact plus its configuration requirements known. The release record should name the source version, migration state, OpenAPI version, smoke requests, logs to watch, rollback trigger, and owner. Have another person execute the checklist before calling it operational.

Add health and readiness routes, graceful signal handling, an example environment file, OpenAPI

output, and a release checklist.

Check your understanding: test and ship

Check handler and integration tests, request IDs and structured logs, configuration, shutdown, versioning, compatibility, and release 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/web-apis/.