Hono Course

Flavio Copes

Updated August 3, 2026

Contents

Hono foundations **2**
- Start from Web Standards . . . 2
- Choose a runtime adapter . . . 2
- Use context deliberately . . . 2
- Return clear responses . . . 3
- Check your understanding: hono foundations . . . 3

Routing and middleware **4**
- Design the route tree . . . 4
- Understand middleware order . . . 4
- Use built-in middleware selectively . . . 4
- Write request context middleware . . . 5
- Check your understanding: routing and middleware . . . 5

Validation and errors **6**
- Validate the correct target . . . 6
- Connect a schema validator . . . 6
- Handle errors centrally . . . 7
- Use cookies and authorization safely . . . 7
- Check your understanding: validation and errors . . . 7

Type-safe clients and testing **8**
- Test with app.request . . . 8
- Use test helpers where they help . . . 8
- Build a typed RPC client . . . 9
- Test the contract, not the types . . . 9
- Check your understanding: type-safe clients and testing . . . 10

Deploy and operate Hono **10**
- Model runtime bindings . . . 10
- Deploy to Node and Workers . . . 10
- Observe portable applications . . . 11
- Finish the portability report . . . 11
- Check your understanding: deploy and operate hono . . . 12

Build a typed Hono API with Web Standard primitives, middleware, validation, testing, inferred clients, and two deployment runtimes.

What you'll learn: Build and test a bookmarks API, then deploy the same portable core to Node and Cloudflare Workers with explicit runtime boundaries.

Hono foundations

Start from Web Standards and separate portable app code from runtime adapters.

Start from Web Standards

Use Request, Response, URL, Headers, and fetch as the mental model beneath Hono.

Before choosing a tool or writing more code, make the requirement concrete. The bookmarks handler reads a standard Request through context and returns a Response-compatible value.

Hono builds around Web Standard request and response APIs, which makes much of the application portable across runtimes. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to learn only framework helper names and ignore the platform objects they wrap. The happy path may still work, which makes the mistake easy to miss. inspect the underlying Request and Response behavior and use helpers where they make intent clearer. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Build the smallest portable Hono application and identify exactly which parts depend on its runtime. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Write one route with Hono helpers and one with an explicit Response, then compare status, headers, and body. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Choose a runtime adapter

Separate portable application code from Node, Workers, Bun, Deno, or another runtime entry point.

Start from the behavior you can observe. One module exports the app; a Node adapter listens on a port; a Worker entry exports fetch behavior.

The Hono app can be shared, but startup, environment bindings, sockets, and some middleware remain runtime-specific. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to assume framework portability means every dependency and API works identically everywhere. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to keep the core on common APIs and isolate adapter, storage, environment, and runtime-specific capabilities. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Build the smallest portable Hono application and identify exactly which parts depend on its runtime. Record enough evidence that another developer can repeat the result without relying on your memory.

Run the same health route under two runtimes and list every file and dependency that differs. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Use context deliberately

Read request data, set response metadata, and pass typed per-request values through Hono context.

Authentication middleware stores a typed user id for later handlers without using a process-global

variable. That contrast gives us something useful to test instead of a rule to memorize.

Context belongs to one request and is a useful carrier for validated state, bindings, and middleware results. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can put request-specific identity or database state in module globals and still get one successful demo. Push past the demo. set narrowly named context variables in middleware and declare their types where the application is created, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Build the smallest portable Hono application and identify exactly which parts depend on its runtime. Save the before-and-after evidence now; it will make the final project review much more honest.

Attach a request id and user object, then prove simultaneous requests do not see each other's values. Save the evidence, then explain which requirement would force you to choose a different design.

Return clear responses

Use text, JSON, redirects, bodies, headers, and status codes as one intentional response contract.

A created bookmark returns JSON, a creation status, and a location header, while deletion returns no invented payload. This is a small part of a typed bookmarks API built with Hono, tested in memory, and deployed once to Node and once to Cloudflare Workers, but the boundary it creates affects everything that follows.

Hono helpers create responses; they do not decide which representation or status is correct for the API. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to return successful JSON for every branch because the helper is convenient. It makes later failures harder to locate. Instead, define route outcomes first, then use the helper that expresses the required status, headers, and body. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Build the smallest portable Hono application and identify exactly which parts depend on its runtime. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Create a response table for success, invalid input, missing bookmark, conflict, and unexpected failure. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: hono foundations

Check Web Standard primitives, adapters, bindings, context, response helpers, and project setup.

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

Take this interactive quiz online

Routing and middleware

Design the route tree and compose typed middleware in a visible order.

Design the route tree

Group resource routes, parameters, methods, and fallbacks into an inspectable API shape.

Before choosing a tool or writing more code, make the requirement concrete. The bookmarks router owns list, create, read, update, and delete paths under `/bookmarks`.

Hono routes match method and path in registration order, and grouped apps can own one coherent prefix. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to scatter resource routes across files with overlapping wildcards and no route table. The happy path may still work, which makes the mistake easy to miss. write the route contract first, group one resource, and test literal, parameter, method, and not-found behavior. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Compose routes and middleware so the request path stays short, typed, and visible. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Build a route table and send requests that differ by method, trailing segment, invalid id, and unknown path. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Understand middleware order

Trace before-next and after-next behavior through nested middleware and handlers.

Start from the behavior you can observe. A timer starts before authentication and records the final status after the route returns.

Hono middleware composes like an onion: code before `await next()` runs inward and code after it runs outward. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to call next without awaiting it or assume after-code runs before the handler. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to diagram the entry and exit order and await next whenever later response state matters. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Compose routes and middleware so the request path stays short, typed, and visible. Record enough evidence that another developer can repeat the result without relying on your memory.

Add three named middleware layers, predict all log lines, and compare the actual order for success and failure. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Use built-in middleware selectively

Apply CORS, secure headers, logging, compression, caching, and other middleware only where their assumptions fit.

The public read API has a narrow CORS policy, while the internal health route does not inherit browser credentials. That contrast gives us something useful to test instead of a rule to memorize.

Built-in middleware saves code but still encodes policy that must match the route and deployment. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can mount every built-in globally to make the app feel production-ready and still get one successful demo. Push past the demo. read the middleware contract, mount it at the smallest scope, and verify headers and behavior with requests, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Compose routes and middleware so the request path stays short, typed, and visible. Save the before-and-after evidence now; it will make the final project review much more honest.

Create allowed and disallowed origin tests and inspect which routes receive each security or caching header. Save the evidence, then explain which requirement would force you to choose a different design.

Write request context middleware

Create one typed middleware that validates an incoming request id or generates a safe one.

The middleware accepts a bounded valid correlation id or creates one, stores it in context, and returns it in the response. This is a small part of a typed bookmarks API built with Hono, tested in memory, and deployed once to Node and once to Cloudflare Workers, but the boundary it creates affects everything that follows.

Good custom middleware has one responsibility, explicit inputs, predictable output, and no hidden global state. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to copy arbitrary unbounded header input into logs and responses. It makes later failures harder to locate. Instead, validate and bound the value, generate a fallback, type the context variable, and test both branches. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Compose routes and middleware so the request path stays short, typed, and visible. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Send missing, valid, oversized, and malformed ids and verify context, response header, and logs. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: routing and middleware

Check route order, grouping, middleware order, context variables, built-ins, and custom composition.

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

Take this interactive quiz online

Validation and errors

Validate the right input target and handle errors, cookies, and authorization.

Validate the correct target

Validate JSON, form, query, header, cookie, or parameter input where it actually enters the route.

Before choosing a tool or writing more code, make the requirement concrete. Bookmark creation validates JSON, search validates query values, and the id parameter becomes a bounded identifier.

Hono validation is target-specific, and content type determines whether body data is parsed as expected. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to read raw values after validation or validate one target while trusting the same name from another. The happy path may still work, which makes the mistake easy to miss. define one authoritative input location, validate it, and use only the validated result downstream. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Reject malformed or unauthorized requests at the boundary and keep domain code independent of raw request parsing. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Send the same field in query, JSON, and a header and prove only the contract location affects the operation. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Connect a schema validator

Use a schema library when it improves reuse and types without hiding the actual HTTP failure contract.

Start from the behavior you can observe. The create schema validates URL and title shape; the service checks whether the normalized URL already exists.

A schema can validate structure and infer types, but the route still owns error status, field reporting, and business rules. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to put database lookups and authorization side effects inside a reusable structural schema. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to separate structural validation from contextual business checks and translate issues into a stable client response. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Reject malformed or unauthorized requests at the boundary and keep domain code independent of raw request parsing. Record enough evidence that another developer can repeat the result without relying on your memory.

Test malformed JSON, invalid URLs, duplicate normalized URLs, and an unknown field according to the documented policy. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Handle errors centrally

Use not-found and error handlers while preserving expected domain outcomes and useful operator context.

Missing bookmarks return a controlled response, while a storage exception reaches onError with a request id and safe body. That contrast gives us something useful to test instead of a rule to memorize.

A centralized error boundary handles unexpected failures; it should not turn every expected outcome into a generic server error. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can throw strings for normal branches or expose full exception details to the client and still get one successful demo. Push past the demo. model expected outcomes explicitly and reserve the central error boundary for unexpected failures, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Reject malformed or unauthorized requests at the boundary and keep domain code independent of raw request parsing. Save the before-and-after evidence now; it will make the final project review much more honest.

Trigger four failure classes and verify status, response shape, logs, and request-id correlation. Save the evidence, then explain which requirement would force you to choose a different design.

Use cookies and authorization safely

Set restrictive cookies and authorize resource access on the server for every protected operation.

An opaque session cookie identifies the user, and the bookmark owner is checked before update or deletion. This is a small part of a typed bookmarks API built with Hono, tested in memory, and deployed once to Node and once to Cloudflare Workers, but the boundary it creates affects everything that follows.

Cookie helpers simplify parsing and serialization; they do not replace session design, CSRF protection, or authorization. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to store trusted roles in a script-readable cookie and skip ownership checks after login. It makes later failures harder to locate. Instead, use secure cookie attributes, keep authority server-side, and check the resource action on every request. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Reject malformed or unauthorized requests at the boundary and keep domain code independent of raw request parsing. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Test another user's bookmark, a missing session, a revoked session, and a state-changing cross-site request. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: validation and errors

Check target-specific validation, typed values, content types, errors, cookies, authorization, and safe output.

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

Take this interactive quiz online

Type-safe clients and testing

Test through app.request and use inferred clients without dropping runtime checks.

Test with app.request

Send Request-compatible inputs directly to the app and assert the full HTTP response.

Before choosing a tool or writing more code, make the requirement concrete. The bookmarks suite sends JSON requests, reads Response objects, and checks status, headers, and parsed bodies.

`app.request()` exercises routing, middleware, validation, handlers, and errors without opening a public listener. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to call handlers directly with invented context mocks. The happy path may still work, which makes the mistake easy to miss. test through app.request and inject controlled storage or environment dependencies at app construction. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Test the real Hono app and use inferred clients without pretending compile-time types validate network data at runtime. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Write success and failure tests for every route, including wrong method, content type, and unknown path. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Use test helpers where they help

Build typed request inputs conveniently without letting helpers hide the contract under test.

Start from the behavior you can observe. A helper creates a typed bookmark request, while the test independently asserts the resulting status and JSON.

Testing helpers reduce request-construction noise, but assertions should still describe observable HTTP behavior. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to assert only that the helper call did not throw. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to use helpers for setup, then inspect the Response and external side effects as a real client would. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Test the real Hono app and use inferred clients without pretending compile-time types validate network data at runtime. Record enough evidence that another developer can repeat the result without relying on your memory.

Repeat one test with a manually constructed Request to confirm the helper has not hidden a header or encoding assumption. Repeat it once with an invalid or hostile condition and write down the

boundary the failure revealed.

Build a typed RPC client

Infer a client from route definitions and understand the repository and compilation boundaries that make it work.

A client in the same workspace gets typed paths, inputs, and outputs from the exported app type. That contrast gives us something useful to test instead of a rule to memorize.

Hono RPC shares TypeScript knowledge between server and client; it is not runtime validation and does not help an unrelated untyped consumer automatically. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can publish a server-only runtime bundle to the browser merely to obtain types and still get one successful demo. Push past the demo. export type information cleanly, keep runtime dependencies separate, and preserve server-side validation, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Test the real Hono app and use inferred clients without pretending compile-time types validate network data at runtime. Save the before-and-after evidence now; it will make the final project review much more honest.

Create a typed client call, then send malformed raw HTTP to prove the server still rejects it. Save the evidence, then explain which requirement would force you to choose a different design.

Test the contract, not the types

Add runtime negative tests, compatibility examples, and response assertions around the inferred client.

The suite verifies error shapes, status codes, headers, pagination boundaries, and unknown fields using raw requests. This is a small part of a typed bookmarks API built with Hono, tested in memory, and deployed once to Node and once to Cloudflare Workers, but the boundary it creates affects everything that follows.

Types catch many development mistakes, but deployed clients can be old, untyped, compromised, or simply wrong. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to delete runtime validation because the trusted TypeScript client compiles. It makes later failures harder to locate. Instead, treat types as developer feedback and HTTP tests as evidence about the deployed boundary. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Test the real Hono app and use inferred clients without pretending compile-time types validate network data at runtime. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Write one test that cannot be expressed as a valid typed client call and verify the server's defensive response. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: type-safe clients and testing

Check app.request, test helpers, dependency injection, RPC clients, shared types, contract scope, and negative tests.

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

Take this interactive quiz online

Deploy and operate Hono

Model bindings, deploy to two runtimes, observe behavior, and measure portability.

Model runtime bindings

Type and inject databases, secrets, caches, and environment configuration according to the target runtime.

Before choosing a tool or writing more code, make the requirement concrete. The Worker receives a database binding through env, while the Node adapter injects a repository built from process configuration.

Bindings are capabilities supplied by the deployment environment and should remain visible in application types and tests. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to read undeclared globals deep inside handlers. The happy path may still work, which makes the mistake easy to miss. define a small environment contract, type it, validate local equivalents, and replace it with fakes in tests. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Deploy the same core app to two runtimes while documenting real differences instead of claiming magical portability. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Run the app with a missing binding and make startup or request failure explicit and testable. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Deploy to Node and Workers

Use the correct adapter and deployment entry point while keeping route code shared.

Start from the behavior you can observe. The Node version owns a listener and shutdown; the Worker version exports fetch and uses platform bindings.

Node and Workers differ in process lifecycle, sockets, filesystem access, environment, and platform limits even when Request and Response match. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to copy the Node entry point into a Worker and patch failures one global at a time. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to isolate runtime entry points, audit dependencies for compatibility, and test both targets

in their native local tools. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Deploy the same core app to two runtimes while documenting real differences instead of claiming magical portability. Record enough evidence that another developer can repeat the result without relying on your memory.

Deploy health and bookmark routes to both targets and record differences in startup, configuration, storage, and logs. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Observe portable applications

Emit request ids, structured errors, durations, and domain metrics through runtime-appropriate sinks.

Both deployments report route, status, duration, and request id without logging bookmark contents or credentials. That contrast gives us something useful to test instead of a rule to memorize.

Portable application logic still needs an observability interface because console and platform tracing differ by runtime. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can couple every handler to one vendor logger or dump full requests to console and still get one successful demo. Push past the demo. define a narrow logging and metrics interface, redact values, and adapt it at the runtime boundary, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Deploy the same core app to two runtimes while documenting real differences instead of claiming magical portability. Save the before-and-after evidence now; it will make the final project review much more honest.

Trigger the same failure in both deployments and trace it from client request id to platform log. Save the evidence, then explain which requirement would force you to choose a different design.

Finish the portability report

Complete the API and write an evidence-based account of what moved unchanged, what was adapted, and what remained platform-specific.

The report lists shared routes and tests, adapter code, binding implementations, unsupported APIs, performance, and operating limits. This is a small part of a typed bookmarks API built with Hono, tested in memory, and deployed once to Node and once to Cloudflare Workers, but the boundary it creates affects everything that follows.

Portability is a measured property of this application and dependency graph, not a slogan attached to the framework. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to declare the application portable after one hello-world route runs twice. It makes later failures harder to locate. Instead, run the complete contract suite on both targets and document every difference that affects development or operations. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Deploy the same core app to two runtimes while documenting real differences instead of claiming magical portability. A short observation with concrete evidence is

more useful than a confident sentence with no reproduction path.

Hand both deployments the same test corpus, compare results, and decide whether the shared core is worth its abstraction cost. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: deploy and operate hono

Check bindings, runtime differences, observability, limits, portability tests, and final deployment evidence.

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/hono/.