

Hono

Version 4.13 · verified 2026-10-26
Fast, lightweight web framework built on Web Standards for any JS runtime.

Contents

Mental model 2
App setup 2
Routing 3
Context & Request 5
Response helpers 6
Middleware & State 8
Error handling 10
Further reading 11

Mental model

Hono is built directly on Web Standard `Request` and `Response` objects (`fetch` API) rather than Node.js HTTP server primitives. An application is a router function that processes an incoming standard `Request` context and returns a Web `Response`. Because it relies solely on portable Web APIs, the exact same code runs without platform adapters on Cloudflare Workers, Deno, Bun, Node.js, and edge networks.

App setup

Runtime	Export / Adapter
Workers	`export default app`
Bun	`export default app`
Node.js	`@hono/node-server`
Deno	`Deno.serve(app.fetch)`

Hono applications are created with `new Hono()`. On Cloudflare Workers or Bun, export the application instance directly as the default export. For Node.js, use

`@hono/node-server` to bind Hono's fetch handler to Node's HTTP server.

```
import { Hono } from "hono";

const app = new Hono();

app.get("/", (c) => {
  return c.text("Hello Hono!");
});

export default app;
```

> **Note:** Cloudflare Workers and Bun handle `app.fetch` natively without extra runtime wrapper packages.

Routing

Pattern	Path example
Named param	`/users/:id`
Optional param	`/posts/:id?`

Pattern	Path example
Wildcard	`/static/*`
Sub-router	`app.route("/api", subApp)`

Routes map HTTP methods and path patterns to handler functions. Sub-applications created as separate Hono instances can be mounted onto specific path prefixes using the `.route()` method for modular organization.

```
import { Hono } from "hono";

const app = new Hono();
const api = new Hono();

api.get("/users/:id", (c) => {
  const id = c.req.param("id");
  return c.text(`User: ${id}`);
});

app.route("/api/v1", api);
```

> **Gotcha:** Dynamic route parameters only match single path segments, not slashes unless wildcard `*` is used.

Context & Request

Accessor	Purpose
`c.req.param("id")`	Read route path parameter
`c.req.query("q")`	Read URL query parameter
`c.req.header("auth")`	Read incoming HTTP header
`c.req.json<T>()`	Parse request JSON body

The context object `c` wraps the incoming Web Standard `Request` alongside response generation utilities. Type-safe accessor methods retrieve path parameters, query params, headers, and parsed payload bodies.

```
import { Hono } from "hono";


type Payload = { name: string };
```

```ts
const app = new Hono();

app.post("/items", async (c) => {
  const query = c.req.query("q");
  const body = await c.req.json<Payload>();
  return c.json({ query, name: body.name });
});
```

> **Warning:** Calling `c.req.json()` on requests with empty or malformed JSON payloads throws a `SyntaxError`.

Response helpers

Method	Output
`c.json(data, status?)`	Application/json response
`c.text(str, status?)`	Text/plain response
`c.html(html, status?)`	Text/html response

Method	Output
`c.redirect(url, status?)`	HTTP redirect response

Response helper methods construct standard Web `Response` instances with appropriate `Content-Type` headers and status codes. Additional custom headers can be set on the context before returning via `c.header()`.

```
import { Hono } from "hono";

const app = new Hono();

app.get("/data", (c) => {
  c.header("X-Custom-Header", "active");
  return c.json({ ok: true }, 200);
});

app.get("/old", (c) => {
```

```
  return c.redirect("/new", 301);
});
```

> **Tip:** Supply status code `201` as the second argument to `c.json()` when returning newly created resources.

Middleware & State

Function	Usage
Global middleware	`app.use("*", middleware)`
Path middleware	`app.use("/api/*", middleware)`
`c.set(key, val)`	Attach state variable
`c.get(key)`	Retrieve state variable

Middleware functions receive context `c` and `next`. Call `await next()` to yield control to downstream handlers. Pass generic `Env` definitions to `new Hono<{ Variables: ... }>()` for type-safe environment and variable access.

```ts
import { Hono } from "hono";

type Env = { Variables: { user: string } };
const app = new Hono<Env>();

app.use("*", async (c, next) => {
  c.set("user", "usr_123");
  await next();
});

app.get("/me", (c) => {
  return c.text(c.get("user"));
});
```

> **Gotcha:** Omitting `await next()` stops middleware execution and prevents downstream route handlers from running.

Error handling

Construct	Purpose
`HTTPException`	Throw structured HTTP error
`app.onError(fn)`	Global exception handler
`app.notFound(fn)`	Custom 404 handler

Throwing `HTTPException` from `hono/http-exception` halts request processing and returns an HTTP error response. Register `app.onError` to intercept unhandled exceptions and produce clean error JSON envelopes.

```
import { Hono } from "hono";
import { HTTPException } from "hono/http-
exception";

const app = new Hono();

app.get("/secure", () => {
  throw new HTTPException(401, {
    message: "Unauthorized",
```

```
  });
});

app.onError((err, c) => {
  return c.json({ error: err.message }, 500);
});
```

> **Warning:** Custom `app.onError` handlers override default logging, hiding stack traces unless explicitly logged.

Further reading

- Hono Documentation
- Hono Routing Guide
- Hono Middleware Overview