Client

August 17, 2026

Client

Overview

Client turns a Hono application schema into a chainable HTTP client at compile time and turns property access on that chain into a request at runtime. `hc` accepts a base URL and optional request options, then returns a `Proxy` cast to the schema-derived `Client<T, Prefix>` type. [src/client/client.ts:133-137; src/client/client.ts:232] The `Client` type walks schema paths into nested property names, with each terminal path mapped to a `ClientRequest` that exposes HTTP-method calls plus `$url` and `$path`. [src/client/types.ts:294-318; src/client/types.ts:67-118]

The runtime proxy collects string property accesses as path segments. Accessing a property creates another proxy; invoking that proxy calls a callback with the accumulated path and invocation arguments. The proxy deliberately returns `undefined` for `then`, preventing a client chain from being treated as a thenable. [src/client/client.ts:15-31]

An endpoint call returns the underlying Fetch `Response`, typed as `ClientResponse` according to the endpoint's declared output, status, and output format. The interface mirrors response body and metadata fields and exposes standard body readers, while its `ok` type narrows to `true` or `false` when the status type is known to be successful or unsuccessful. [src/client/types.ts:120-154] `parseResponse` is the optional consuming layer: it accepts either a response or a response promise, parses its body, and rejects non-OK responses with `DetailedError`. [src/client/utils.ts:84-114; src/client/fetch-result-please.ts:14-44]

How it works

From chain access to a request

When an invoked path ends with a dollar-prefixed segment, `hc` strips the prefix to obtain the method name; all earlier segments are joined with slashes and merged with the base URL. [src/client/client.ts:161-170] For ordinary method calls, the callback creates `ClientRequestImpl` with that URL, method, and the selected query serializer. [src/client/client.ts:214-229]

`ClientRequestImpl.fetch` accepts Hono validation-target-shaped arguments with optional query, form, JSON, parameter, header, and cookie inputs. [src/client/client.ts:53-58] Query input is passed to the configured search-parameter builder. [src/client/client.ts:59-62] The default builder creates `URLSearchParams`, skips `undefined` values, appends every array element under the same key, and sets scalar values. [src/client/utils.ts:26-44]

Form input becomes a `FormData` body. Undefined form entries are skipped, while array values are appended once per element. [src/client/client.ts:64-79] JSON input is serialized with `JSON.stringify` and sets `Content-Type` to `application/json`. [src/client/client.ts:81-84] Path parameters are retained for URL substitution. [src/client/client.ts:86-88]

The request combines argument headers with option headers. Option headers may be a record or a synchronous or asynchronous function; the function result is awaited before `Headers` is constructed. [src/client/client.ts:93-96; src/client/types.ts:57-65] Cookie arguments are serialized individually through the shared cookie serializer and joined into the `Cookie` header. [src/client/client.ts:98-104] A JSON body's content type is then assigned to that header map. [src/client/client.ts:106-110]

Before dispatch, the implementation removes a terminal `/index` representation, substitutes declared path tokens, and appends the serialized query string when one exists. [src/client/client.ts:111-118] `replaceUrlParam` recognizes `:name`, optional parameters, and parameter segments with an attached brace expression; a missing parameter value removes the matched segment. [src/client/utils.ts:18-24] `removeIndexString` treats an absolute root `/index` specially by retaining its slash, while other terminal `/index` text is removed. [src/client/utils.ts:55-60]

The method is uppercased. GET and HEAD calls omit the constructed body; other methods pass it to Fetch. [src/client/client.ts:91-92; src/client/client.ts:119-128] Dispatch uses the per-request `fetch` option when present and otherwise the ambient `fetch`. [src/client/client.ts:122-128] `init` is spread after the generated body, method, and headers, so fields in `init` can replace those generated values. [src/client/client.ts:122-128; src/client/types.ts:35-41]

Base options and per-call options are recursively merged before dispatch. When both layers contain headers, `hc` replaces the per-call headers with an async function that combines base headers first and request headers afterward; `deepMerge` recursively merges object-valued properties and otherwise takes the source value. [src/client/client.ts:217-229; src/client/utils.ts:62-82]

URL and path jobs

A chain ending in `$url` does not issue a request. It constructs the merged URL, optionally substitutes parameters and appends query data, removes the index representation, and returns `new URL(result)`. [src/client/client.ts:169-185] `$path` follows the same construction but removes the normalized base URL and returns a leading-slash path. [src/client/client.ts:171-186]

At the type level, `$url` can return a `TypedURL` when the base prefix is a literal absolute URL. `TypedURL` narrows `protocol`, host fields, pathname, search, origin, and `href` from the supplied prefix, route path, parameter argument, and query argument; a nonliteral prefix instead yields `URL`. [src/client/types.ts:168-193; src/client/types.ts:215-230] `$path` is typed as a template-literal path whose parameter placeholders can be resolved when their values are string literals, and whose query component becomes `?${string}` when a nonempty query input exists. [src/client/types.ts:156-166; src/client/types.ts:194-213]

WebSocket jobs

A chain ending in `$ws` takes a separate branch. It substitutes path parameters, changes an HTTP-style protocol to its WebSocket counterpart, constructs a `URL`, and writes query values into `targetUrl.searchParams`; array query values are appended individually. [src/client/client.ts:187-203; src/client/utils.ts:46-53] It calls an optional `webSocket` constructor from client options when supplied, otherwise it instantiates the ambient `WebSocket`. [src/client/client.ts:204-211; src/client/types.ts:32-34] The `$ws` member exists in the client type only when the route's GET schema declares `outputFormat: 'ws'`. [src/client/types.ts:112-118]

Response consumption and failure behavior

`parseResponse` delegates directly to `fetchRP`. [src/client/utils.ts:92-114] `fetchRP` awaits the response, checks for a body or polyfill `_bodyInit`, and does not consume bodies for the listed no-body response statuses. [src/client/fetch-result-please.ts:7; src/client/fetch-result-please.ts:14-31] For other bodies, it chooses `json()` only when

the content type matches the JSON media-type expression, including structured JSON suffixes; absent or other content types use `text()`. [src/client/fetch-result-please.ts:77-95]

After parsing, a non-OK response causes `fetchRP` to throw `DetailedError` with a message composed from status and status text, a `statusCode`, and detail containing the parsed data and status text. [src/client/fetch-result-please.ts:33-40] `DetailedError` extends `Error` and records optional `detail`, `code`, `log`, and `statusCode` fields. [src/client/fetch-result-please.ts:46-75] Transport failures from Fetch are not caught or wrapped by this path; the test suite distinguishes a failed Fetch from an HTTP response parsed by `parseResponse`. [src/client/fetch-result-please.ts:14-44; src/client/utils.test.ts:305-321]

The return type of `parseResponse` filters client response unions to content-bearing statuses that are not client or server error statuses, then infers JSON output, declared string output, or `string`; if no qualifying response remains, it resolves to `undefined`. [src/client/utils.ts:92-111; src/utils/http-status.ts:6-72] This is a type-level filter only: at runtime, any non-OK Fetch response still throws `DetailedError`. [src/client/utils.ts:92-114; src/client/fetch-result-please.ts:33-44]

Configuration

`hc` reads client-level `headers`, `fetch`, `webSocket`, `init`, and `buildSearchParams` from `ClientRequestOptions`. [src/client/types.ts:32-65] Per-call options use the same option type and are merged with the client-level options before the Fetch call. [src/client/types.ts:234-237; src/client/client.ts:217-229]

`buildSearchParams` is the customization point for query serialization. If no custom function is passed, `hc` uses the local default builder; if one is passed, both request dispatch and `$url` / `$path` generation use that function. [src/client/client.ts:137-139; src/client/client.ts:171-180; src/client/client.ts:214-216] The default convention emits repeated keys for array values. [src/client/utils.ts:26-44]

A custom `fetch` can be either the platform Fetch function or Hono's request method type, which permits in-process request execution rather than a network call. [src/client/types.ts:28-34] The testing helper applies that option by wrapping `app.request` and handing the wrapper to `hc` with a localhost base URL. [src/helper/testing/index.ts:16-27]

Wiring

The public client module exports `hc`, `parseResponse`, `DetailedError`, and the request, response, inference, and schema-transformation types. [src/client/index.ts:6-17] The root package separately re-exports request and response inference types together with `ClientRequestOptions`. [src/index.ts:46-48]

Runtime request construction depends on the shared cookie serializer for cookie header values and on browser-compatible globals including `FormData`, `Headers`, `URLSearchParams`, Fetch, `URL`, and, on the WebSocket branch, `WebSocket`. [src/client/client.ts:3; src/client/client.ts:65-78; src/client/client.ts:110; src/client/client.ts:122-128; src/client/client.ts:192-211] Response typing depends on Hono schema and endpoint types, router method constants, HTTP status types, and utility types imported from adjacent framework modules. [src/client/types.ts:1-6]

The schema boundary is explicit in the `Client` type: it accepts a `HonoBase`, extracts its schema, and transforms route strings into the nested client chain. [src/client/types.ts:311-318] `ApplyGlobalResponse` and `PickResponseByStatusCode` operate on an app type by rewriting or filtering its extracted endpoint schema, allowing the resulting app type to be passed to `hc`. [src/client/types.ts:332-363; src/client/types.ts:365-393]

Outside this directory, the testing helper consumes `hc` and its client option types to create an in-process client, while static-site generation imports only `replaceUrlParam` for route substitution. [src/helper/testing/index.ts:6-27; src/helper/ssg/ssg.ts:1-15]

25 entities in `src/client`. **1 other subsystem depends on it**, which makes it the 5th most depended-upon part of this codebase.

What it is made of

Its 25 entities sit in 3 files under `src/client`: 13 type aliases, 8 functions, 3 interfaces and 1 class.
`types.ts` holds 16 of them — more than any other file here.
`ClientResponse` declares 8 methods, the widest surface here.

Where work enters

- `BuildSearchParamsFn` — `src/client/types.ts`:30
- `ClientRequestOptions` — `src/client/types.ts`:32
- `ClientRequest` — `src/client/types.ts`:67
- `ClientResponse` — `src/client/types.ts`:128

Boundaries

1 other subsystem depends on this one — `Helper`. Changing what it exposes changes them.

They hold 1 edge into it between them. What they reach is narrower than the folder: 1 of its 25 members carries every inbound edge — `replaceUrlParam` (1).

It depends on no other subsystem in this repository — it is a leaf.