Model Context Protocol (MCP)

Version spec 2026-07-28 · TS SDK 2.4 · verified 2026-11-02
Build MCP servers in TypeScript — transports, tools, resources and prompts, schema design, OAuth, registering in Claude Code, testing, security.

Contents

Mental model 2
Hosts, clients and servers 2
Transports 3
Tools, resources and prompts 4
Designing tool inputs and results 6
Serving over HTTP 7
Authorization 9
Registering a server in Claude Code 10
Testing 11
Security 12
Further reading 13

Mental model

MCP is a JSON-RPC contract that lets any AI application discover a server's tools, resources and prompts at runtime and call them, so one integration works in every compatible client. The server never talks to the model: it answers requests from a client, and the host application decides what the model sees and what it may run. Since revision 2026-07-28 the protocol is stateless — every request carries its own protocol version and capabilities, with no handshake and no session.

Hosts, clients and servers

Role	Is	Example
Host	The AI application the user runs	Claude Code, an IDE
Client	One connection inside the host	One per server
Server	Exposes tools, resources, prompts	Your `notes` server

A host creates one client per server and keeps servers isolated from each other. Messages are JSON-RPC 2.0 in UTF-8. Revision 2026-07-28 removed the `initialize` handshake: each request carries `io.modelcontextprotocol/protocolVersion` and the client's capabilities in `_meta`, and servers must implement `server/discover` so a client can ask up front which versions they speak.

```
{
  "jsonrpc": "2.0", "id": 1,
  "method": "tools/call",
```

```
  "params": {
    "name": "get_order",
    "arguments": { "id": "ord_1" },
    "_meta": {
      "io.modelcontextprotocol/protocolVersion":
        "2026-07-28"
    }
  }
}
```

> **Note:** Earlier revisions (2025-11-25 and before) used an `initialize` handshake and, over HTTP, an `Mcp-Session-Id` header. The spec calls those “legacy”; SDK v2 serves both eras.

Transports

Transport	Use for	Wire
stdio	Local server the host launches	One JSON message per line
Streamable HTTP	Remote server, many clients	POST per message; JSON or SSE reply
HTTP+SSE	Nothing new	Deprecated

With stdio the host starts your server as a subprocess and speaks over stdin/stdout. With Streamable HTTP every client message is a separate `POST` to one endpoint, and the server replies with a JSON object or an

SSE stream scoped to that request. Protocol semantics are identical on both.

```
import { McpServer }
  from "@modelcontextprotocol/server";
import { StdioServerTransport }
  from "@modelcontextprotocol/server/stdio";


const server = new McpServer({
  name: "notes",
  version: "1.0.0",
});
// register tools, resources, prompts, then:
const transport = new StdioServerTransport();
await server.connect(transport);
console.error("notes server ready"); // stderr
```

> **Gotcha:** On stdio, stdout *is* the protocol. One stray `console.log` writes a line that is not valid MCP and the client drops or fails the connection. Log to stderr.

Tools, resources and prompts

Primitive	Controlled by	Typical use
Tool	The model	Actions: query, create, send
Resource	The application	Context: files, config, schemas

Primitive	Controlled by	Typical use
Prompt	The user	Templates, slash commands

The split is about who decides to use it. The model chooses to call a tool; the host chooses which resources to attach; the user picks a prompt from a menu. Expose read-only context as a resource when the host should control it, and as a tool when the model should fetch it on demand.

```
import { McpServer }
  from "@modelcontextprotocol/server";


const server = new McpServer({
  name: "workspace",
  version: "1.0.0",
});
server.registerResource(
  "config", "config://app",
  { mimeType: "text/plain" },
  async (uri) => ({
    contents: [{ uri: uri.href, text: "eu" }],
  }),
);
```

> **Tip:** Revision 2026-07-28 deprecates the Roots, Sampling and Logging features. Pass paths as tool arguments, call your LLM provider directly, and log to stderr or OpenTelemetry instead.

Designing tool inputs and results

Field	Purpose
`description`	What it does and when to use it
`inputSchema`	JSON Schema of the arguments
`outputSchema`	Shape of `structuredContent`
`annotations`	Hints such as `readOnlyHint`
`isError: true`	Failure the model can read and fix

The model picks and fills tools from the name, description and schema alone, so treat them as the prompt. Constrain arguments (enums, regex, ranges) and describe each field. Tool names should be 1–128 characters of letters, digits, `_`, `-` and `.`, unique per server. Return execution failures as a result with `isError: true` and a message that says what to try next; reserve JSON-RPC errors for unknown tools and malformed requests.

```
import { McpServer }
  from "@modelcontextprotocol/server";
import * as z from "zod/v4";
const s = new McpServer({ name: "shop",
  version: "1" });
```

```
s.registerTool("get_order", {
  description: "One order by ID. Read-only.",
  inputSchema: z.object({ id: z.string()
    .regex(/^ord_\w+$/).describe("ord_<n>") }),
  annotations: { readOnlyHint: true },
}, async ({ id }) => id === "ord_1"
  ? { content: [{ type: "text", text: "42" }] }
  : { isError: true, content: [{ type: "text",
    text: "Not found. Try search_orders" }] });
```

> **Gotcha:** A handler that throws does not crash the call — the SDK turns the exception's `message` into an `isError` result, which can leak internals to the model. Catch and compose the message yourself.

Serving over HTTP

Piece	Job
`createMcpHandler(factory)`	Web-standard `fetch` handler
`toNodeHandler`	Adapts it to `node:http`
Host/Origin validation	Blocks DNS rebinding
Bind `127.0.0.1`	Never `0.0.0.0` for local servers

In SDK v2 the factory runs once per HTTP request and returns a fresh `McpServer`, so the endpoint holds no state and scales horizontally. Create connection pools at module scope and close over them. The handler validates nothing itself: mount Host and Origin checks in front

(the Express, Fastify and Hono adapters arm them by default on localhost).

```
import { createServer } from "node:http";
import { createMcpHandler, McpServer }
  from "@modelcontextprotocol/server";
import * as mw
  from "@modelcontextprotocol/node";

const handler = createMcpHandler(() =>
  new McpServer({ name: "kb", version: "1" }));
const node = mw.toNodeHandler(handler);
const host = mw.localhostHostValidation();
const origin = mw.localhostOriginValidation();
createServer((q, s) => {
  if (!host(q, s) || !origin(q, s)) return;
  void node(q, s);
}).listen(3000, "127.0.0.1");
```

> **Gotcha:** There is no `Mcp-Session-Id` any more. State you need across calls goes in an explicit, server-minted handle passed as an ordinary tool argument — never on the server instance.

Authorization

Requirement	Source
Server is an OAuth 2.1 resource server	OAuth 2.1 draft
Publish protected resource metadata	RFC 9728
Client sends `resource` = server URI	RFC 8707
Server checks token audience is itself	RFC 8707
Client registers by metadata document	CIMD (DCR deprecated)

Authorization is optional and applies to HTTP; stdio servers read credentials from the environment. A protected server answers an unauthenticated request with `401` and a `WWW-Authenticate: Bearer` challenge whose `resource_metadata` points at its metadata document, served under `/.well-known/oauth-protected-resource`. That document names the authorization server. The client then runs authorization code + PKCE, sending the `resource` parameter so the token is minted for this server only.

```
{
  "resource": "https://mcp.example.com/mcp",
  "authorization_servers": [
    "https://auth.example.com"
  ],
  "scopes_supported": ["files:read"]
}
```

> **Warning:** Never forward the client's token to an upstream API (token passthrough is forbidden). The server must reject tokens not issued for it, and call upstream APIs with its own separate token.

Registering a server in Claude Code

Scope	Stored in	Shared
`local` (default)	`~/.claude.json`, per project	No
`project`	`.mcp.json` in the repo	Yes, via git
`user`	`~/.claude.json`	No, all projects

Everything after `--` is the command that launches a stdio server. Remote servers take `--transport http` and a URL; Claude Code handles the OAuth flow. Tools appear to the model as `mcp__<server>__<tool>`.

```
claude mcp add --transport stdio notes \
  -- node build/server.js
claude mcp add --scope project \
  --transport http docs \
  https://mcp.example.com/mcp
claude mcp list
```

> **Gotcha:** Interactive sessions ask before loading a repo's `.mcp.json` servers, but `claude -p` and Agent SDK runs load them without asking. Untrusted repos in CI get their servers started.

Testing

Tool	Use
`npx @modelcontextprotocol/inspector <cmd>`	Web UI against a stdio server
`... --cli <cmd> --method tools/list`	Scriptable checks in CI
`... --tui <cmd>`	Terminal UI
In-process `Client`	Unit tests without a port

The Inspector needs Node 22.19 or newer. For automated tests, drive the real handler through a real `Client` whose transport calls `handler.fetch` directly — no socket, no mock.

```
import { Client, StreamableHTTPClientTransport }
  from "@modelcontextprotocol/client";
import { createMcpHandler, McpServer }
  from "@modelcontextprotocol/server";

const h = createMcpHandler(() =>
  new McpServer({ name: "kb", version: "1" }));
const t = new StreamableHTTPClientTransport(
  new URL("http://test.local/mcp"),
  { fetch: (u, i) =>
    h.fetch(new Request(u, i)) });
const client = new Client({ name: "t",
  version: "1" });
```

```
await client.connect(t);
await client.listTools(); // { tools: [] }
```

Security

Threat	Control
Injected instructions in tool results	Treat results as data; human approval
Over-broad tools	Narrow tools, least-privilege tokens
Lying annotations	Trust hints only from trusted servers
Token theft or replay	Audience checks, short-lived tokens
DNS rebinding	Host/Origin checks, loopback bind

Every tool result lands in the model's context, so a server that fetches web pages, issues or email can smuggle instructions to the model (indirect prompt injection). The spec says there should always be a human able to deny tool calls. Design tools so the worst injected call is cheap: read-only by default, scoped credentials, no "run anything" tools.

```
bad:  run_sql(query: string)
good: get_order(id), list_orders(status, limit)
```

> **Warning:** `readOnlyHint` and `destructiveHint` are claims, not guarantees. Clients must treat annotations as untrusted unless the server itself is trusted.

Further reading

- MCP specification 2026-07-28
- Key changes in 2026-07-28
- MCP authorization
- MCP TypeScript SDK
- MCP Inspector
- Claude Code MCP guide