MCP Course

Flavio Copes

Updated August 6, 2026

Contents

Learn how MCP connects AI applications to tools and data, then configure, inspect, and safely use an existing server.

What you'll learn: Understand the MCP roles and primitives, connect a server you trust, inspect its capabilities, and verify its access boundaries.

Understand MCP

Learn the protocol roles, primitives, lifecycle, and where MCP fits.

What MCP solves

Understand why a standard connection between AI applications and external capabilities is useful before writing any server code.

Model Context Protocol, or MCP, is a standard way for an AI application to discover and use capabilities exposed by another program.

Without a shared protocol, every application needs a custom integration for every data source or action. MCP gives both sides a common vocabulary for discovery, input schemas, calls, results, and errors.

Think of it as a boundary between two programs:

Diagram: MCP integration boundary. An AI host uses an MCP client to call an

MCP server, which reaches a narrowly scoped file system, database, or API with its own credentials. The host can connect to a new server without learning that server's private implementation. The server can expose the same capability to more than one compatible host. This reduces integration work, but only when the capability has a clear contract.

MCP does not make an integration trustworthy by itself. A server is still software with access to data and systems. You must understand what it can do and give it only the access it needs.

It also does not replace the underlying API, database driver, or permission system. A GitHub server may still call GitHub's API. MCP describes how the host discovers and invokes the server-facing operation.

Start from a user task, not from "connect everything." A useful project-notes server might expose search and read operations over one folder. It does not need a general shell command or access to the whole disk.

In this course we will first connect one existing server. Then we will build a small project-notes server with two read-only tools, one resource, and one prompt.

Draw one integration you use today. Mark the host, server, underlying system, credentials, and point where a human approves an action. If those boundaries are unclear, the integration is not ready to trust.

Hosts, clients, and servers

Separate the three MCP roles so you know where the model, connection, and external capability actually live.

The **host** is the AI application the person uses. It owns the interface, model interaction, permissions, and the overall user experience.

An MCP **client** is the connection inside the host. It speaks the protocol to one MCP server.

The **server** exposes a focused set of capabilities. It might read project notes, query a database, or call an existing API.

One host can manage several clients, each connected to a different server. The server does not become the model, and the model does not run inside the protocol.

This separation helps diagnose failures. If the server process crashes, the host interface and model can still work, but that client's capabilities disappear. If one client cannot initialize, another server connection may remain healthy. If the model proposes a bad call, the host can still require approval before the client sends it.

The host owns policy such as which servers are enabled, which capabilities are shown to the model, and when the person must approve. The client owns one protocol session, including initialization and negotiated features. The server owns the schemas and implementation of its capabilities.

Credentials should follow the same boundary. A database password needed by one server should go to that server process, not into the model prompt or every other server's environment.

Draw the path for this course: `AI host → MCP client → project-notes server → notes data`. Now label where an invalid tool argument is rejected, where filesystem access is restricted, and where the person sees an approval.

When debugging, name the failing role instead of saying "MCP is broken." That one distinction usually cuts the search space dramatically.

MCP, APIs, and function calling

See MCP as an integration layer around useful application capabilities rather than a replacement for APIs or model tool calling.

An HTTP API defines how software talks to a service. Function calling lets a model produce arguments for a declared function. MCP connects these ideas across hosts and servers.

An MCP server can call an API you already have. It can turn a broad API into a smaller interface with names, descriptions, schemas, results, and permission boundaries that make sense to an AI application.

For example, a billing API may expose dozens of endpoints. An MCP server could expose one `find_invoice` tool that accepts an invoice number and returns only the fields needed for support. The server handles API authentication and translates upstream failures into a useful result. The model never needs the raw API token.

The host still decides whether and when to present a capability to a model. The server executes only the operations it exposes.

Do not mirror every API endpoint automatically. Start from a real user task and expose the smallest operation that completes it.

The layers have different responsibilities:

- the API defines the service's native operations and authorization
- the MCP server adapts selected operations into focused capabilities
- the protocol handles discovery and messages
- model tool calling proposes structured arguments
- the host decides what reaches the model and what requires approval

Schemas improve structure, not truth. A string that matches an `invoiceId` schema may still name an invoice the current user cannot access. The server must authenticate, authorize, validate, rate-limit, and log sensitive actions just like any other application boundary.

Take one existing API endpoint and design a smaller task-level tool around it. List what the tool deliberately omits, which identity it acts as, and which upstream errors callers need to distinguish.

MCP complements APIs and function calling; it does not erase their security or product-design work.

Tools, resources, and prompts

Choose the MCP primitive that matches an action, readable context, or reusable message workflow.

A **tool** is an operation a client can call with arguments. Searching notes is a tool because the caller supplies a query and receives a computed result.

A **resource** is readable context identified by a URI. A catalog of available notes works as a resource because the client can list and read it.

A **prompt** is a reusable message template. A project-review prompt can accept a topic and produce a consistent review request.

Use the primitive that makes the contract honest:

- `delete_note(id)` is a tool because it performs an action and needs explicit arguments.
- `notes://projects/alpha/readme` is a resource because it identifies readable context.
- `review_project(topic)` can be a prompt because it assembles reusable guidance for a conversation.

A resource is not automatically public or safe. Its URI still needs access checks, and resource contents can contain untrusted instructions. A prompt is not trusted code; it is content the host may present or combine with other messages. A tool result can also contain hostile or malformed external data.

The boundaries matter. Do not hide a state-changing action inside a resource read, and do not use a prompt as a substitute for data validation.

Names and descriptions shape how the model chooses a capability. Make them specific, state side effects, and use narrow input schemas. `run_action` with a free-form string is much harder to review than `archive_note` with a note ID and a documented result.

Classify five operations from an application you know. For each one, explain whether the client should call, read, or present it and where authorization belongs.

Tools act, resources expose addressable context, and prompts package a reusable conversational workflow.

Requests, discovery, and protocol versions

Follow discovery and one capability call while keeping the negotiated protocol revision separate from your application code.

A client first discovers what a server offers. It can list tools, resources, and prompts, then call or read one of them using the declared contract.

Discovery is a runtime exchange, not a promise that every connected server has the same surface forever. A host can initialize a connection, learn the server's advertised capabilities, list a category, and then make a request using one returned name and schema.

MCP uses JSON-RPC messages, but the SDK handles the wire details. Your application code registers capabilities and returns their results.

The protocol changes over time. A revision identifies the contract a peer supports. Current servers should document which SDK and protocol revision they target.

Keep three versions separate:

- the MCP protocol revision negotiated by peers
- the SDK package version used to implement a client or server
- your server application's own release version

Upgrading an SDK can change application-facing APIs without changing your capability names. Changing the protocol can affect negotiation and wire behavior. Changing your server can alter tool semantics even when both other versions stay fixed.

This course targets the TypeScript SDK v2 line and the final 2026-07-28 protocol. The v2 server package is still marked beta while its API settles, so pin and review the version before production use.

Record all three versions in diagnostics. Test initialization, discovery, one valid call, one invalid argument, and an older supported peer before upgrading production. Avoid writing business logic against raw JSON-RPC details when the SDK already owns them; keep your capability handlers separable so an SDK migration does not rewrite the domain code.

Inspect one server's startup or debug output and identify its protocol, SDK, and application versions. If one is absent, decide where you would expose it safely.

Check your understanding: MCP foundations

Check the roles, primitives, API relationship, discovery flow, and reason protocol revisions matter.

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

Take this interactive quiz online

Connect an existing server

Choose, configure, test, and troubleshoot a server you trust.

Choose a server you trust

Evaluate an existing server as installed software before giving it access to local files, accounts, or credentials.

Start with a server from a publisher you can identify. Read its source when available, recent release history, installation command, and requested permissions.

Ask four questions: What data can it read? What can it change? Where does it send data? Which credentials does it receive?

Also inspect the supply chain. A command such as `npx -y package-name` can download and execute the latest matching package under your user account. Pin a reviewed version when reproducibility matters, verify the actual registry owner and repository, and treat install scripts as executable code.

Permissions include more than configured folders. The process may inherit environment variables, network access, the current working directory, and operating-system privileges. A read-only tool schema does not prove the server process itself cannot write files.

For this exercise, choose a read-only server with a small scope. A local filesystem server limited to a disposable practice folder is easier to review than an account-wide administration server.

Create a temporary folder with two harmless text files. Do not point the exercise at your home directory, SSH keys, password stores, or real customer data.

Write a small trust record:

- exact package and version
- publisher and source repository
- command that runs
- filesystem and network scope
- credentials supplied
- capabilities advertised

- update and removal plan

Then run the server in the smallest practical operating-system and host scope. If a server requires broad access for a narrow task, choose another implementation or do not install it.

Repeat the review after upgrades. Trust applies to the code version you examined, not permanently to a package name.

Read a server configuration

Understand the name, command, arguments, environment, and transport before copying a configuration into an AI host.

A local stdio configuration tells the host which process to start. The important fields are the server name, command, arguments, and optional environment variables.

Here is the shape, using placeholders rather than a real secret:

```
{
  "mcpServers": {
    "practice-notes": {
      "command": "npx",
      "args": ["-y", "example-package", "/absolute/path/to/practice"],
      "env": {
        "API_TOKEN": "<API_TOKEN>"
      }
    }
  }
}
```

The exact configuration file location depends on the host. Use its current documentation instead of assuming every host reads the same file.

Read the configuration as a process launch, not as passive data. In the example, the host runs `npx` under your account, allows it to download a package because of `-y`, passes a filesystem path, and injects an environment value. Changing any argument can change the server's authority.

Prefer an absolute executable path or a well-understood runtime resolution when the host has a different `$PATH` from your terminal. Quote and encode arguments through the configuration format; do not build one shell string unless the host explicitly requires a shell.

Never commit a real token. Prefer the host secret store or an environment variable when it supports one.

An environment-variable placeholder is only safe if something outside the checked-in file resolves it. Confirm the host's semantics: some pass the literal placeholder, some expand from their environment, and some provide a separate secret setting.

Before installing, translate the JSON into one sentence: “The host starts this exact executable and package, grants this directory, and supplies these variables.” Run the command manually in the disposable environment and check its version and help output.

Finally, keep a copy of the last working configuration without secrets. That gives you a reviewable rollback when an upgrade changes the launch contract.

Generate and install the configuration

Build a syntactically valid local configuration without placing secret values in the generated JSON.

Open the MCP Config Generator. Choose your host and the stdio transport.

Enter a short server name, the command, and each command argument. Use an absolute path for the disposable practice folder.

If the server needs a credential, enter only the environment-variable name. The generator deliberately uses a placeholder instead of collecting the value.

Copy the JSON to the configuration location documented by your host. Restart or reload the host so it reads the new server.

Validate the generated shape before installation:

- the server name is unique and recognizable
- command and arguments are separate values
- the package version is pinned when appropriate
- the practice path is absolute and narrow
- no secret value appears in JSON
- the selected transport matches the server

Make a backup of the working host configuration. Add only this server, reload, and inspect the host's diagnostics. If the whole configuration fails to parse, restore the backup rather than making several speculative edits.

Confirm that the server appears by name and that its advertised capabilities match what you expected.

Discovery is part of installation verification. A process that starts but advertises an unexpected write tool or broader resource scope has not passed review. Disable it and investigate the exact installed version.

Configuration controls how the host launches the server; it does not sandbox the process by itself. Keep using a disposable folder and harmless data for the first call.

Save the sanitized configuration and the expected capability list. You can compare them after an upgrade instead of relying on memory.

Make a controlled first call

Test one read-only operation against disposable data and inspect the proposed arguments before approving the call.

Ask the host to read or search one file in the practice folder. Use a request whose expected result you already know.

If the host shows an approval screen, inspect the server name, tool name, and arguments. Approval is meaningful only when you read what will run.

Compare the result with the source file. Then ask for a path outside the allowed practice folder and confirm the request is rejected.

Test the contract in layers:

1. Make one valid call with a known result.

2. Send a missing or wrong-type argument and inspect the validation error.
3. Request a nonexistent item and distinguish "not found" from a server crash.
4. Attempt the known permission boundary.
5. Confirm the server wrote nothing during read-only tests.

A friendly host response is not enough evidence. Check the server log, the disposable directory, and the exact returned content. A model can summarize a result incorrectly even when the underlying tool call succeeded.

Approval should describe the real effect at a useful level. If the screen shows only an opaque tool name or a giant unreviewable payload, improve the server name, description, and arguments before trusting higher-impact operations.

Record the server version, configuration scope, successful call, and rejected boundary. This becomes a small trust test you can repeat after an upgrade.

Do not move to real data until both the intended capability and denied boundary behave as designed. A positive test without a negative test proves very little about scope.

Troubleshoot a connection

Diagnose a server that fails to start, disappears, or corrupts the stdio stream without weakening its permissions.

Run the exact configured command in a terminal. A missing executable, invalid package name, bad path, or absent environment variable usually appears immediately.

Diagnose in order:

1. Can the host parse its configuration?
2. Can it find and start the executable?
3. Does the process stay alive long enough to initialize?
4. Can client and server agree on protocol behavior?
5. Does discovery return the expected capabilities?
6. Does one minimal call work?

This turns "the server disappeared" into a specific failed boundary.

For a stdio server, stdout is the protocol. Startup banners and debug logs on stdout can break the connection. Send diagnostic text to stderr instead.

Run the configured command with the same working directory and environment the host uses. A command that succeeds in your interactive shell may depend on aliases, a version manager, or variables the desktop host never inherits. Use absolute paths temporarily to prove resolution, then choose a maintainable configuration.

Check the host log for the server process exit code and stderr. Then change one thing and retry the same discovery request.

If initialization fails after an upgrade, compare the pinned SDK, protocol revision, server arguments, and advertised capability schema with the last working record. Roll back one version rather than widening access or reinstalling unrelated components.

Do not solve a path error by granting access to the entire filesystem. Fix the path or configuration while keeping the smallest useful scope.

Write down the failed layer, evidence, one change, and retest result. That small discipline preserves the clues that random configuration changes erase.

Check your understanding: connect a server

Check server trust, configuration fields, secret handling, stdio logs, controlled calls, and troubleshooting.

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/mcp/.