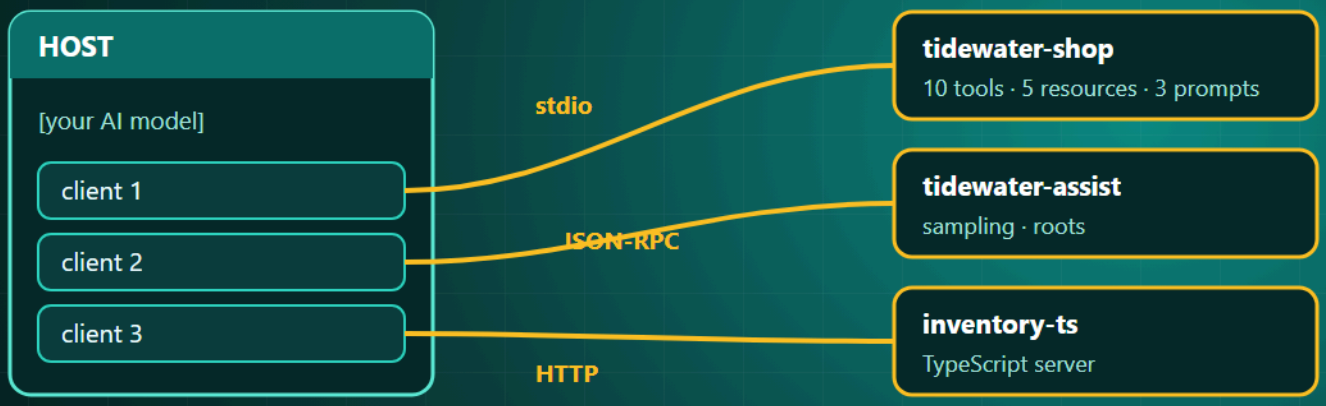



```python
@mcp.tool(annotations=READ_ONLY)
def check_stock(sku: str)
      -> StockLevel:
  """Stock level of one SKU."""
  return store.check_stock(sku)
```

```
{"method": "tools/call",
 "isError": false, "status": "low"}
```

REPAIR QUOTE

TUNE-BASIC

Labor	$79.00
Parts	$53.50
Tax (parts)	$4.01
Total	**$136.51**

A COMPLETE, PRACTICAL GUIDE

Model Context Protocol in Practice

Build, test, secure, and ship MCP servers and clients with Python and TypeScript

Tools · Resources · Prompts · Elicitation · JSON-RPC
stdio · Streamable HTTP · OAuth · Testing · Deployment

SERVERS · CLIENTS · SECURITY · PRODUCTION

THENEXBOOKS

Model Context Protocol in Practice

Build, Test, Secure, and Ship MCP Servers and Clients with Python and TypeScript
From your first stdio tool to an authorized, observable, production deployment — built step by step

First Edition

THENEXBOOKS

Model Context Protocol in Practice
Build, Test, Secure, and Ship MCP Servers and Clients with Python and TypeScript

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. The Model Context Protocol specification, its software development kits, and the applications that use them change frequently; protocol revisions add, deprecate, and remove features, and SDK APIs can change between releases. Always consult the official specification and SDK documentation for the versions you use, and review the security implications before connecting any tool to an AI system or exposing a server on a network.

Python is a trademark of the Python Software Foundation. TypeScript, Windows, and Visual Studio Code are trademarks of Microsoft Corporation. Node.js is a trademark of the OpenJS Foundation. OpenTelemetry is a trademark of The Linux Foundation. Other product and company names mentioned are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the maintainers of the Model Context Protocol or by any AI company.

Tidewater Bike Co., the bike shop used as the running example in this book, is fictional, as are its staff, customers, products, repair tickets, bookings, authorization server, and all sample data. Email addresses use the reserved example.com domain and phone numbers use the fictional 555 range. Any resemblance to real businesses or people is coincidental.

Preface

Every useful AI assistant eventually needs to reach outside its own conversation. It needs to look up a stock level, open a ticket, read a policy document, or book an appointment. For a while, every application solved that problem its own way: one plugin format for this assistant, another integration for that editor, a custom wrapper for each internal API. Connecting M applications to N systems meant writing something close to M × N integrations, and none of them could be reused.

The Model Context Protocol, or MCP, is an open protocol that replaces those one-off integrations with a single contract. A *server* describes the tools, resources, and prompts it offers; a *client* inside an AI application — the *host* — discovers and uses them through a small set of JSON-RPC messages over stdio or HTTP. Build a server once and any MCP-capable host can use it. Build a client once and it can talk to any MCP server.

This book teaches MCP the way you will actually use it: by building, testing, securing, and shipping real servers and clients. We do not stop at a "hello world" tool. We build one realistic system step by step, read the actual messages that cross the wire, and explain *why* the protocol works the way it does, so you can design your own servers with confidence and keep up as the specification evolves.

Who This Book Is For

- **Python developers** who want to give AI assistants safe, well-designed access to their own data and systems.
- **Backend and platform engineers** who need to expose internal APIs to AI tools, run MCP servers over HTTP, and put authorization, logging, and deployment around them.
- **Developers building AI applications** who want to write MCP clients, connect several servers to one host, and drive tool calls from a model.
- **Tech leads and security-minded engineers** who need to understand the risks — prompt injection through tools, over-privileged tokens, untrusted servers — and how to design against them.

You should be comfortable with basic Python: functions, classes, type hints, and a little `async` / `await`. No prior MCP experience is assumed. One chapter uses TypeScript; basic JavaScript knowledge is enough to follow it.

What You Will Learn

- The MCP architecture — hosts, clients, and servers — and the building blocks: tools, resources, prompts, elicitation, sampling, and roots.
- How the protocol works on the wire: JSON-RPC requests, responses, and notifications, version negotiation, and the stdio and Streamable HTTP transports.

- How to build servers with the Python SDK: tool schemas from type hints, structured output, annotations, errors, resources and templates, prompts, completions, progress, and lifespan.
- How to write clients, connect servers to hosts and models, ask the user for confirmation with elicitation, and borrow the client's model and workspace with sampling and roots.
- How the handshake-era revisions and the 2026-07-28 revision differ, and how to build servers and clients that work with both.
- How to serve MCP over HTTP, build a server with the TypeScript SDK, test servers with in-memory clients and pytest, and debug with wire logs and the MCP Inspector.
- How to protect remote servers with OAuth-based authorization, defend against prompt injection and tool poisoning, and design tools with least privilege and confirmation.
- How to package, deploy, and observe servers in production with logs, OpenTelemetry traces, and metrics.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains what MCP is, its building blocks, the JSON-RPC messages and lifecycle, the transports, and how to set up the project. **Part II — Building Servers in Python** (Chapters 6–10) builds the Tidewater server: tools, resources, prompts, completions, context, and progress. **Part III — Clients and Interaction** (Chapters 11–15) writes clients, connects servers to hosts and models, covers elicitation, sampling, and roots, and explains protocol versions and compatibility. **Part IV — HTTP, TypeScript, and Testing** (Chapters 16–20) serves MCP over HTTP, builds a TypeScript server, tests and debugs servers, and combines many servers. **Part V — Security and Authorization** (Chapters 21–24) covers OAuth-based authorization, protecting a real server, the threat landscape, and safe tool design. **Part VI — Production** (Chapters 25–28) covers packaging, deployment, observability, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter includes hands-on examples, "Try It Yourself" exercises, and a **Key Takeaways** box for quick review.

The Running Example: Tidewater Bike Co.

Throughout the book we build the MCP servers for **Tidewater Bike Co.**, a fictional bike shop on Harbor Street in the fictional town of Port Wren. The shop sells parts and accessories, repairs bikes in two service bays, and books customers into service slots. Noor Saleem, the shop's developer, wants staff to be able to ask an AI assistant questions like "Do we have an 11-speed chain in stock?", "What would a basic tune-up with a new chain and shift cables cost?", or "Book Anika's gravel bike into the first free slot on Tuesday" — and have the assistant answer from the shop's real data, safely.

We start with a single read-only tool over stdio and grow it into `tidewater-shop`: ten tools, five resources, and three prompts, served over stdio and HTTP, protected with bearer tokens, tested in both protocol eras, and instrumented with logs and traces. Along the way we add a small assistant

server for sampling and roots, a TypeScript mirror of the inventory tools, and a client that ties it all together. The sample data is small enough to check by hand — a repair quote of $136.51 is labor, parts, and 7.5% tax on parts that you can verify with a calculator.

Versions and Conventions

MCP is versioned by dated specification revisions. This book follows the **2026-07-28** revision, the newest one the SDKs used here implement, and also teaches the **handshake-era** revisions (2025-11-25 and earlier), which many hosts and servers still speak. Where a detail differs between the two, the text says which revision it applies to. The Python examples use the **Python SDK, version 2.x**, and were run on Python 3.12; the TypeScript chapter uses the **TypeScript SDK v2** on Node.js 22. Check your host's documentation for the revision it supports, and the official specification at modelcontextprotocol.io for the current state of any feature.

The messages shown in this book were captured from the book's own sample code. Long strings in those captures are shortened with "..." so that they fit the page. Where we could not run something ourselves — a particular host application, the MCP Inspector, or a cloud deployment — the output is labeled as illustrative. AI models appear as the placeholder *[your AI model]*, because the protocol works the same whichever model your host uses.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause data loss, errors, or security problems.

Code, commands, JSON messages, and file names appear in `monospace`. Commands are shown for Windows with the `py` launcher (for example `py -3.12 -m tidewater_mcp.server`); on macOS and Linux use `python3` instead. Files are named relative to the project root, such as `tidewater_mcp/server.py`.

Contents

Preface

PART I — FOUNDATIONS

1 What MCP Is and Why It Exists: Hosts, Clients, and Servers

2 The Building Blocks: Tools, Resources, Prompts, and Client Features

3 Under the Hood: JSON-RPC Messages and the Protocol Lifecycle

4 Transports: stdio and Streamable HTTP

5 Setting Up: Python, the MCP SDK, and the Tidewater Project

PART II — BUILDING SERVERS IN PYTHON

6 Your First Server: tidewater-shop over stdio

7 Designing Tools: Schemas, Structured Output, Annotations, and Errors

8 Resources and Resource Templates

9 Prompts and Completions

10 Context, Progress, Lifespan, and Change Notifications

PART III — CLIENTS AND INTERACTION

11 Writing an MCP Client in Python

12 Connecting Servers to Hosts and Models: Configuration and the Tool-Use Loop

13 Elicitation: Asking the User for Input and Confirmation

14 Sampling and Roots: Borrowing the Client's Model and Workspace

15 Protocol Versions and Compatibility: The Handshake Era and the 2026-07-28 Revision

PART IV — HTTP, TYPESCRIPT, AND TESTING

16 Streamable HTTP Servers in Python

17 Building a Server with the TypeScript SDK

18 Testing MCP Servers: In-Memory Clients, pytest, and Contract Tests

19 Debugging: Wire Logs, the MCP Inspector, and Common Failures

20 Working with Many Servers: Session Groups, Namespacing, and Gateways

PART V — SECURITY AND AUTHORIZATION

21 Authorization Concepts: OAuth 2.1 for MCP

22 Protecting tidewater-shop: Bearer Tokens, Scopes, and Protected Resource Metadata

23 Threats: Prompt Injection Through Tools, Tool Poisoning, and Confused Deputies

24 Safe Tool Design: Least Privilege, Confirmation, and Guardrails

PART VI — PRODUCTION

25 Packaging and Distributing MCP Servers

26 Deploying Remote Servers: Hosting, Scaling, and Statelessness

27 Observability: Logging, Tracing, and Metrics

28 Case Study: The Complete Tidewater MCP Platform

APPENDICES

A MCP Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What the Model Context Protocol is and the problem it solves, the roles of hosts, clients, and servers, the building blocks every server and client shares, the JSON-RPC messages and lifecycle underneath, the stdio and Streamable HTTP transports, and setting up the Tidewater Bike Co. project.

1 · What MCP Is and Why It Exists

2 · The Building Blocks

3 · JSON-RPC Messages and the Lifecycle

4 · Transports: stdio and Streamable HTTP

5 · Setting Up the Tidewater Project

CHAPTER 1 · PART I — FOUNDATIONS

What MCP Is and Why It Exists: Hosts, Clients, and Servers

An AI assistant is only as useful as the systems it can reach. In this chapter we look at the integration problem the Model Context Protocol solves, meet the three roles every MCP conversation involves, and read our first real `tools/call` exchange, captured from the code you will build in this book.

The Problem: Every Assistant Needs Every System

Large language models are good at reading, writing, and reasoning over text, but on their own they know nothing about your business. They cannot see today's inventory, they do not know which repair tickets are waiting for parts, and they cannot book a service appointment. To be useful at work, an assistant needs *context* (data it can read) and *capabilities* (actions it can take). Both live in other systems: databases, ticket trackers, calendars, file shares, internal APIs.

For a long time, every team that wanted to connect an assistant to one of those systems wrote custom glue. A chat application would grow a plugin format, an IDE assistant would grow a different extension format, and a command-line coding agent would grow a third. Each integration had to be written again for each application. If there are M AI applications and N systems you want them to reach, you end up maintaining something close to $M \times N$ integrations, each with its own conventions for describing actions, passing arguments, reporting errors, and handling credentials.

```
Without a shared protocol (M x N custom integrations)

  Desktop assistant ----+------+------+----> Inventory DB
                        \      \      \
  IDE assistant ---------+------+------+----> Ticket tracker
                         \      \      \
  Coding agent ----------+------+------+---> Booking calendar

  3 applications x 3 systems = 9 pieces of glue code, all different
```

The Model Context Protocol (MCP) attacks this problem the same way earlier standards attacked similar ones: it defines one shared language between the two sides. An AI application that speaks MCP can talk to any system that speaks MCP. A system wrapped once as an MCP server can be used by any AI application that speaks MCP. The $M \times N$ problem becomes an $M + N$ problem: each application implements the client side once, and each system implements the server side once.

```
With MCP (M + N implementations)

  Desktop assistant --+                   +-- tidewater-shop (inventory, tickets)
                      |                   |
  IDE assistant ------+---- MCP ---------- +-- calendar server
                      |                   |
  Coding agent -------+                   +-- file server
```

This is not a new idea. Language servers solved a similar problem for code editors: instead of every editor implementing every programming language, editors speak one protocol and each language ships one server. MCP borrows that shape (and, as we will see in Chapter 3, the same JSON-RPC message format) and applies it to the connection between AI applications and the tools and data they need.

What MCP Standardizes

It helps to be precise about what the protocol actually covers. MCP is an open specification that defines:

- **A message format.** Every message is a JSON-RPC 2.0 request, response, or notification. Chapter 3 takes the format apart message by message.
- **A small set of building blocks.** Servers can offer *tools* (actions a model can call), *resources* (data an application can read), and *prompts* (templates a user can pick). Clients can offer features back to servers, such as *elicitation* (asking the user a question). Chapter 2 covers each one.
- **How the two sides discover each other's features.** Each side advertises *capabilities*, so a client knows whether a server has tools at all, and a server knows whether a client can answer questions.
- **Transports.** The same messages can travel over a local subprocess's standard input and output (*stdio*) or over HTTP (*Streamable HTTP*). Chapter 4 covers both.
- **Authorization for remote servers.** When a server is reached over HTTP, the specification describes how clients obtain and present access tokens. Part V of the book is devoted to it.

Notice what is *not* on that list: which model you use, how the model decides to call a tool, what your user interface looks like, or how you store your data. MCP is deliberately narrow. It is the plug and socket, not the appliance.

The Three Roles: Hosts, Clients, and Servers

Every MCP setup involves three roles. The names are easy to mix up at first, so we will be careful with them throughout the book.

The host

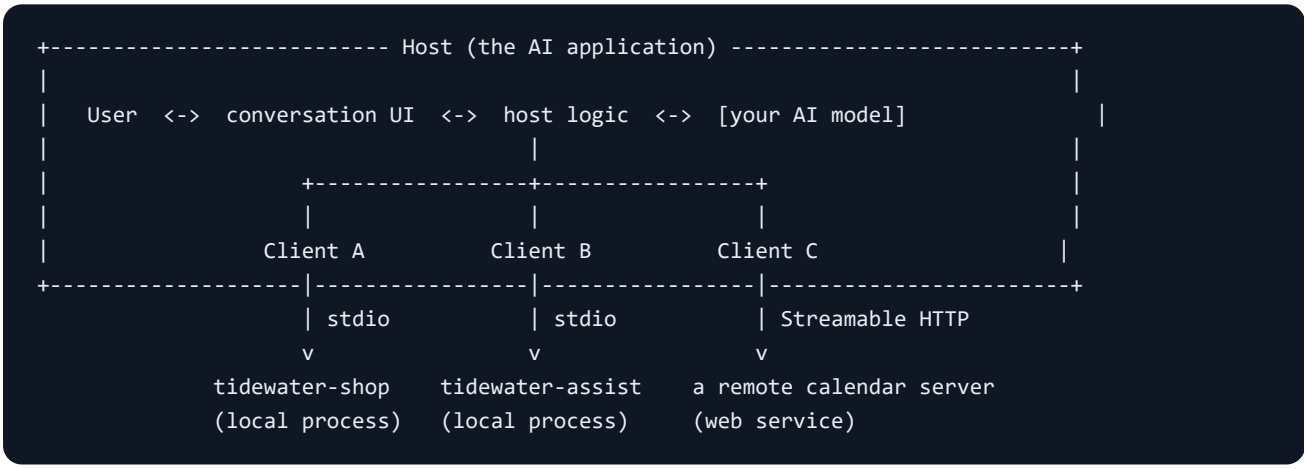
The **host** is the AI application the user actually works with: a desktop AI assistant, an IDE assistant, a coding agent, or a custom internal app you build yourself. The host owns the conversation with the user, talks to the language model, decides which servers to connect to, and enforces the user's choices about what is allowed. When the model says "I want to call `check_stock` with SKU `TW-CHAIN-11`", it is the host that decides whether that call actually happens.

The client

The **client** is the protocol component inside the host that speaks MCP to one server. This is the detail people most often get wrong: **a host creates one client per server connection**. A host connected to three servers runs three clients, each with its own connection, its own negotiated protocol version, and its own view of that server's capabilities. The client is not something the user sees; it is a library object inside the host. In this book, the Python SDK's `Client` class plays that role.

The server

The **server** is a program that exposes some system through MCP. It might run as a local process that the host starts (a stdio server), or as a long-running web service that many hosts connect to (a Streamable HTTP server). A server declares what it offers, answers requests, and enforces its own rules. The server we build in this book is called `tidewater-shop`, and it exposes a bike shop's inventory, repair tickets, quotes, and service-bay bookings.



Role	What it is	Responsibilities
Host	The AI application	User interface, model calls, consent, connecting to servers, combining results
Client	Protocol component inside the host	One connection to one server: negotiation, requests, responses, notifications
Server	A program exposing a system	Declares tools, resources, and prompts; executes requests; enforces business rules

Why the one-client-per-server rule? Isolation. Each server is a separate trust domain. The inventory server should not see what the calendar server returned, and a misbehaving server should not be able to interfere with another server's connection. Keeping one client per connection gives the host a clean place to apply per-server policy: which tools are allowed, which need confirmation, which credentials go where. We will rely on that separation heavily in Part V.

> **NOTE**
>
> The word "client" is overloaded in everyday speech. In this book, "client" always means the MCP protocol component, never "the customer" and never "the browser". When we mean the whole application, we say "host". Tidewater's customers are always called customers.

What MCP Is Not

Because MCP sits next to some very hyped technology, it is easy to expect too much or too little from it. A few clarifications will save you confusion later.

MCP is not a model

No intelligence lives in the protocol. A server does not need a language model to work: `tidewater-shop` is ordinary Python code over a SQLite database, and every answer it gives is deterministic. The model lives in the host. Throughout this book we write "[your AI model]" wherever a real model would sit, and our host examples use a small stub function in its place, so every example runs without network access or an account.

MCP is not an agent framework

MCP does not decide which tool to call, how to plan a multi-step task, when to stop, or how to recover from failure. Those are agent-design questions that live in the host (or in whatever framework the host uses). MCP only standardizes how the host *discovers* and *invokes* capabilities. Other books cover agent design and retrieval pipelines in depth; here we stay focused on the protocol and on building servers and clients that use it well.

MCP is not a replacement for your HTTP APIs

Your existing REST or GraphQL APIs keep serving your web and mobile apps. An MCP server is usually a thin adapter in front of the same business logic, shaped for a different consumer: a model that reads tool descriptions in natural language and fills in arguments from a conversation. The design rules are different, too. A good REST API might expose fifty fine-grained endpoints; a good MCP server usually exposes a smaller set of task-shaped tools with clear descriptions, because every tool definition costs the model attention. Chapter 7 returns to this.

MCP is not a security boundary by itself

The protocol gives you places to put security (authorization for HTTP servers, capability negotiation, confirmations through elicitation), but connecting a server to a host does not make anything safe automatically. Tool output can carry prompt-injection text, tool descriptions can lie, and a local server runs with the user's own permissions. Part V treats these threats seriously.

> **WARNING**
> A local MCP server is just a program running on your computer with your user account's rights. Installing one is equivalent to running its code. Only connect servers you trust, read what a server's tools can do before you enable it, and prefer servers whose source you can inspect.

Where MCP Fits Next to Function Calling

If you have used a model API with "function calling" or "tool use", you already know half of the story. With function calling, your application sends the model a list of function definitions (a name, a description, and a JSON Schema for the arguments). When the model wants to use one, it returns a structured request instead of plain text. Your application runs the function and sends the result back to the model, which continues the conversation.

Function calling answers the question "how does the model ask for an action?" It does not answer "where do the function definitions come from, and who runs them?" Without MCP, the application author hard-codes both. With MCP, the host asks each connected server for its tool list, translates those tool definitions into whatever format its model expects, and routes the model's calls back to the right server.



Look at who talks to whom. The model never talks to the server directly. The host sits in the middle of every exchange, which is exactly what makes approval prompts, logging, and policy enforcement possible. In Chapter 12 we build this loop ourselves, with a function named `run_turn` and a stub model, and we write the small converter that turns MCP tool definitions into function-calling definitions.

The two ideas are complementary. Function calling is the model-facing half; MCP is the system-facing half. A host that supports MCP still uses function calling (or something equivalent) with its model. What MCP adds is a standard way to plug new capabilities in without changing the host's code.

The Open Specification and Its SDKs

MCP is defined by an open specification published at `modelcontextprotocol.io`, together with official software development kits (SDKs) for several languages. The specification is versioned by date. Each revision is identified by a string such as `2025-11-25`, and both sides of a connection agree on one revision before they do any real work.

The SDK we use in this book knows five revisions: `2024-11-05`, `2025-03-26`, `2025-06-18`, `2025-11-25`, and `2026-07-28`. They fall into two groups that behave quite differently on the wire, and you will meet both throughout the book:

- **The handshake era (`2025-11-25` and earlier).** A connection starts with an `initialize` request, in which the client and server agree on a version and exchange capabilities, followed by a `notifications/initialized` message. Over HTTP, the server hands out a session ID. Most hosts and servers deployed before mid-2026 speak one of these revisions.
- **The `2026-07-28` revision.** There is no handshake and no protocol session. A client can ask `server/discover` what a server supports, and every request carries its own protocol version, client information, and client capabilities in a `_meta` field. When a server needs something from the user in the middle of a call, it returns an `input_required` result and the client retries the request with the answer.

We teach the two side by side because you will meet both in practice. A server you write today may be used by a host that only speaks an older revision, and a client you write may connect to servers that speak either one. The good news is that the SDKs handle most of the difference for you. The Python SDK's `Client`, in its default `"auto"` mode, first probes a server with `server/discover` and falls back to the `initialize` handshake when the server is older; a server built with the SDK answers both styles. Chapter 15 explains the differences in depth.

We use two SDKs:

- **The Python SDK, version 2.x** (the package is called `mcp`). Its server class is `MCPServer` and its client class is `Client`. Almost all of our code uses it.
- **The TypeScript SDK, version 2**, which is split into packages such as `@modelcontextprotocol/server`. Chapter 17 rebuilds part of our server with it.

> **NOTE**
>
> MCP is evolving quickly. Revisions are added, features are deprecated, and hosts adopt new revisions at different speeds. Whenever this book describes something that differs between revisions, it says which revision it means, for example "in the 2026-07-28 revision" or "in handshake-era revisions". Check your host's documentation for which revision it speaks, and check the specification at `modelcontextprotocol.io` for anything that has changed since this book was written.

Meet Tidewater Bike Co.

Protocols are easier to learn with a concrete system in front of you, so this book follows one small business from start to finish. Tidewater Bike Co. is a fictional bike shop at 212 Harbor Street in the fictional town of Port Wren. It is open Tuesday to Saturday from 09:00 to 18:00 and Sunday from 10:00 to 16:00, and it is closed on Mondays. Repairs happen in two service bays, B1 and B2.

The people you will meet:

- **Marisol Avery**, the owner, who wants staff to spend less time looking things up.
- **Gideon Okonkwo**, the service manager, who plans each day's repairs.
- **Ivy Castellan** and **Callum Ostrowski**, the mechanics.
- **Bea Thornquist** at the front desk, who answers the phone, writes up repairs, and books appointments.
- **Noor Saleem**, the shop's developer, and our point-of-view character. Noor builds the MCP servers in this book.

The shop runs on a small database: eight inventory items (from an 8.99 inner tube to a 749.00 city bike), a service menu of five repair types, five customers, five repair tickets in various states, and a booking calendar for the two bays. The data is fictional, small enough to fit on a page, and re-seeded every time the server starts, so every example in the book prints the same numbers on your machine as it does in the text.

What Noor wants to build

Marisol's request is simple: staff should be able to ask an AI assistant ordinary questions and get answers from the shop's real data. Bea should be able to type "Can we fit a flat repair in tomorrow morning?" and have the assistant check open slots. Gideon should be able to ask "What is waiting for parts?" Ivy should be able to say "Quote a basic tune-up with a new chain and shift cables" and get a correct total with tax.

Noor could build a custom chat app with all of this hard-wired. Instead, Noor decides to expose the shop's systems through MCP. That way the shop can use whichever MCP host suits each person (a desktop assistant at the front desk, an IDE assistant on Noor's own machine) and switch hosts later

without rewriting the integration. The same server will also serve a small custom host that Noor writes in Chapter 12.

By the end of the book, the Tidewater platform consists of:

- `tidewater-shop`, the main server, written in Python, with ten tools (such as `check_stock`, `quote_repair`, and `book_service`), five resources, and three prompts. It runs over stdio for local use and over Streamable HTTP for remote use, optionally behind token-based authorization.
- `tidewater-assist`, a small server that demonstrates the client features sampling and roots.
- `tidewater-inventory-ts`, a TypeScript server that mirrors the inventory tools.
- A demo client, a small host loop with a tool policy, a gateway, tests, packaging, and deployment notes.

Here is the worked example you will see most often. A customer wants a basic tune-up (service code `TUNE-BASIC`, 79.00 labor) plus a new 11-speed chain (39.50) and a shift cable kit (14.00). Parts total 53.50. Tidewater charges 7.5% tax on parts only, so tax is 4.0125, rounded half-up to 4.01, and the total is **$136.51**. When the `quote_repair` tool returns that number, you will know everything is wired up correctly.

A First Look at a Real Exchange

Before we discuss any more theory, let us look at what actually travels between a client and a server. The book's sample code includes wire captures: real traffic recorded from the canonical Tidewater code by a small pass-through logger. Each line is one JSON-RPC message, prefixed with its direction, `C->S` (client to server) or `S->C` (server to client). We pretty-print the JSON and shorten long strings with "..." to fit the page; the originals are one line per message.

This capture comes from a handshake-era connection over stdio. First, the client introduces itself:

wire/stdio-legacy.log (excerpt)

```
C->S {
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-11-25",
    "capabilities": {"elicitation": {"form": {}, "url": {}}},
    "clientInfo": {"name": "tidewater-client", "version": "1.0.0"}
  }
}
```

The client says which revision it wants to speak, which optional features it supports (here, it can answer elicitation questions in both form and URL mode), and who it is. The server answers:

```
S->C {
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "capabilities": {
      "prompts": {"listChanged": false},
      "resources": {"listChanged": false, "subscribe": false},
      "tools": {"listChanged": false}
    },
    "instructions": "Tools for Tidewater Bike Co. staff: ...",
    "protocolVersion": "2025-11-25",
    "serverInfo": {"name": "tidewater-shop", "version": "1.0.0"}
  }
}
```

The server agrees on the revision, says it has prompts, resources, and tools, offers a paragraph of `instructions` the host may show to the model, and identifies itself. The client confirms with a notification (a message without an `id`, which never gets a reply):

```
C->S {"jsonrpc": "2.0", "method": "notifications/initialized"}
```

Now the real work can begin. Here is the heart of MCP: a `tools/call` request asking for the stock level of the 11-speed chain, and the server's result.

```
C->S {
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {"name": "check_stock", "arguments": {"sku": "TW-CHAIN-11"}}
}
S->C {
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "content": [
      {"type": "text", "text": "{\n  \"sku\": \"TW-CHAIN-11\", ..."}
    ],
    "isError": false,
    "structuredContent": {
      "sku": "TW-CHAIN-11",
      "name": "11-Speed Chain",
      "in_stock": 3,
      "reorder_level": 5,
      "status": "low"
    }
  }
}
```

A few things are worth noticing already, and we will return to each of them in later chapters:

- The request names a tool and passes its arguments as a JSON object. The server published the shape of those arguments earlier, as a JSON Schema in its `tools/list` response.
- The response `id` matches the request `id`. That is how the client pairs answers with questions, even when several requests are in flight.
- The result carries the same data twice: as human- and model-readable text in `content`, and as machine-readable JSON in `structuredContent`. Chapter 7 explains why.
- Three chains are in stock and the reorder level is five, so the status is `low`. That is a business rule in the shop's code, not something the protocol knows about.

What happens when something goes wrong? Here is the same tool called with a SKU that does not exist:

```
S->C {
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "content": [
      {"type": "text",
       "text": "Error executing tool check_stock: Unknown SKU: TW-NOPE"}
    ],
    "isError": true
  }
}
```

The tool failed, but the *protocol* did not: the server still sent a normal `result`, with `isError` set to `true` and an explanation in plain text. That design is intentional. The error text goes back to the model, which can read it, realize it used a wrong SKU, and try again, perhaps by calling `search_inventory` first. Protocol-level errors (a malformed message, an unknown method) are a different thing, and Chapter 3 shows how they look.

> **TIP**
>
> Get into the habit of reading wire traffic early. When an integration misbehaves, the messages on the wire are the ground truth: they show exactly what the host asked for and exactly what the server answered. Chapter 4 shows how to capture your own traffic with the book's `wiretap.py` logger, and Chapter 19 builds a full debugging toolbox around it.

The same call in the 2026-07-28 revision

For comparison, here is the beginning of a capture from the same code when both sides speak the `2026-07-28` revision. There is no `initialize`. The client asks the server what it supports, and every request carries the client's details in `_meta`:

wire/stdio-modern.log (excerpt)

```
C->S {
  "jsonrpc": "2.0",
  "id": 1,
  "method": "server/discover",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "tidewater-client", "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {
        "elicitation": {"form": {}, "url": {}}
      }
    }
  }
}
```

The `check_stock` result in this revision has the same `content`, `isError`, and `structuredContent`, plus a `resultType` of `"complete"` and the server's identity in `_meta`. The core idea (name a tool, pass arguments, get content back) is unchanged. What changed is the scaffolding around it, and that is exactly the kind of detail the SDK hides from you most of the time.

Running the Demo Client

You do not have to take the captures on trust. The book's sample code includes a demo client that starts `tidewater-shop` as a subprocess, talks to it over stdio, and prints a short report. Chapter 5 walks you through setting up Python and the project; if you already have Python 3.12 and the `mcp` package, you can run it now from the project folder. On Windows we use the `py` launcher; on macOS and Linux, replace `py -3.12` with `python3`.

```
py -3.12 -m tidewater_mcp.client --yes
```

Output

```
protocol 2026-07-28; server tidewater-shop
tools: search_inventory, check_stock, quote_repair, get_repair_ticket,
       create_repair_ticket, update_ticket_status, find_open_slots, book_service,
       audit_stock, cancel_booking
quote: labor 79.00 parts 53.50 tax 4.01 total 136.51
open slots 2026-11-03: 8
# Tidewater Bike Co. service menu
prompt messages: 1
[elicitation] Cancel booking BK-502? The slot will be released.
cancel_booking: ok {"booking_id": "BK-502", "ticket_id": "RT-1002",
                    "slot_id": "2026-11-03-1300-B2", "status": "cancelled"}
```