

A TECHNICAL BOOK

MCP Apps for Claude

Build interactive UI inside the conversation

Sandeep Kumar Patel

September 2026

COPYRIGHT

MCP Apps for Claude *Build interactive UI inside the conversation*

Copyright © 2026 Sandeep Kumar Patel. All rights reserved. No part of this book may be reproduced or transmitted in any form without prior written permission of the author, except for brief quotations in a review. The companion code is released under the MIT License; see [LICENSE](#) in the code download.

The information here is provided "as is," without warranty. MCP Apps is a young extension and Claude changes often: every version, screen, setting, and observed behavior was accurate when checked in September 2026, and may not be now. Neither the author nor any distributor is liable for loss or damage caused by this book's contents.

"Claude" is a trademark of Anthropic, PBC. This book is independent and is not affiliated with, sponsored by, or endorsed by Anthropic. "MCP" originates with Anthropic, PBC. "ChatGPT" is a trademark of OpenAI and "Cloudflare" of Cloudflare, Inc. Other trademarks belong to their owners.

Found a stale fact, a bug, or an error? Report it at tutorialsavvy.com. First edition, 2026.

Contents

Preface

Conventions Used in This Book

PART I: GETTING STARTED

Chapter 1: What Are MCP Apps?

- 1.1 From Text to Interfaces
- 1.2 One Feature, Two Names
- 1.3 When an App Is Worth It
- 1.4 Where Apps Run
- 1.5 Claude Stays in Control
- 1.6 Summary
- 1.7 Exercises

Chapter 2: MCP Basics

- 2.1 Getting Started with Snug
- 2.2 Tools and Their Results
- 2.3 Annotations and Permissions
- 2.4 Connecting over HTTP or stdio
- 2.5 Choosing the Right SDK Setup
- 2.6 Always Return Useful Text
- 2.7 Summary
- 2.8 Exercises

Chapter 3: Building Your First App

- 3.1 How an App Fits Together
- 3.2 The Server Side
- 3.3 The Client Side
- 3.4 Testing Locally with basic-host
- 3.5 Running Your App in claude.ai
- 3.6 Running Your App in Claude Desktop
- 3.7 Four Common Mistakes
- 3.8 Summary
- 3.9 Exercises

PART II: DESIGN AND LAYOUT

Chapter 4: Layouts: Inline, Carousel, and Fullscreen

- 4.1 Rules for Inline Cards
- 4.2 Building a Carousel
- 4.3 Building a Fullscreen View
- 4.4 Switching Display Modes in Code
- 4.5 App or Chat?
- 4.6 Summary
- 4.7 Exercises

Chapter 5: Theming Your App

- 5.1 Start with a Transparent Background
- 5.2 Applying Claude's Styles
- 5.3 Using Claude's Style Variables
- 5.4 Loading Claude's Font
- 5.5 Borders and Safe Areas
- 5.6 Dark Mode and Accessibility
- 5.7 Summary
- 5.8 Exercises

Chapter 6: Designing for Mobile

- 6.1 How Phones Are Different
- 6.2 What Phones Can't Do
- 6.3 Let the Conversation Scroll
- 6.4 Tap Targets
- 6.5 Handling Safe Areas on Phones
- 6.6 What to Check on a Phone
- 6.7 Snug on an Android Phone
- 6.8 Summary
- 6.9 Exercises

PART III: HOW YOUR APP BEHAVES

Chapter 7: The App Lifecycle

- 7.1 The Lifecycle, Step by Step
- 7.2 What Claude Actually Sends
- 7.3 Loading and Error States
- 7.4 The Invisible App
- 7.5 When a Card Is Removed
- 7.6 Summary
- 7.7 Exercises

Chapter 8: Talking to Claude

- 8.1 Three Ways to Talk
- 8.2 Suggested Messages Wait for the Person
- 8.3 Keeping Claude Up to Date
- 8.4 Tools Only the Card Can Call
- 8.5 Opening Links
- 8.6 File Downloads
- 8.7 Summary
- 8.8 Exercises

Chapter 9: Handling Multiple Copies of a Card

- 9.1 Every Call Creates a New Card
- 9.2 Deciding Which Card Is Newest
- 9.3 The Election
- 9.4 What an Older Card Does
- 9.5 Where Your Cards Live
- 9.6 Summary
- 9.7 Exercises

Chapter 10: Large and Live Data

- 10.1 How Big Can a Result Be?
- 10.2 Splitting Data into Pages
- 10.3 Keeping Data Up to Date
- 10.4 Moving Large Files
- 10.5 Summary
- 10.6 Exercises

PART IV: SECURITY

Chapter 11: Sandboxing and Content Security

- 11.1 How the Sandbox Works
- 11.2 Declaring What Your Card Needs
- 11.3 The Policy the Host Builds
- 11.4 Checking What Your App Actually Loads
- 11.5 Two Kinds of Request
- 11.6 Frames and Device Permissions
- 11.7 Summary
- 11.8 Exercises

Chapter 12: Authentication and Security

- 12.1 How Claude Asks for Permission
- 12.2 Sign-In Only When Needed
- 12.3 Knowing Who's Calling
- 12.4 Asking for More Permissions
- 12.5 Handling Other People's Text
- 12.6 A Threat Model for Snug
- 12.7 Summary
- 12.8 Exercises

PART V: TESTING AND PUBLISHING

Chapter 13: Testing and Debugging

- 13.1 Six Ways to Test
- 13.2 Testing the Server
- 13.3 Make Sure Your Tests Can Fail
- 13.4 Testing the Card Without a Browser
- 13.5 Testing in basic-host
- 13.6 Debugging in Claude
- 13.7 Screenshots as Tests
- 13.8 Checking the Submission Rules Automatically
- 13.9 Before Every Release
- 13.10 Summary
- 13.11 Exercises

Chapter 14: Publishing to the Directory

- 14.1 Three Ways to Share Your App
- 14.2 What the Reviewers Check
- 14.3 Is Booking Destructive?
- 14.4 Allowed Links

- 14.5 The Listing and Screenshots
- 14.6 After Publishing
- 14.7 Getting Snug Ready for Production
- 14.8 Summary
- 14.9 Exercises

Appendix A: MCP Apps Message Reference

Appendix B: Snug Checklist

Appendix C: Beyond Claude

Appendix D: Exercise Solutions

Appendix E: Troubleshooting

Appendix F: MCP Apps at a Glance

Glossary

About the Author

Colophon

Preface

Somewhere in a conversation with Claude, a small piece of your own software appears: a row of meeting rooms, a form, a chart. You touch it, and it answers. That's an MCP App. This book shows you how to build one that Claude accepts, that looks as if it belongs in the conversation, and that keeps working when Claude is on a phone, when the person is signed out, when a second copy appears, and when a reviewer opens it.

The plan is simple. You'll build one connector, called Snug, that books meeting rooms, one small step per chapter. And you'll run every step against Claude itself, instead of trusting what the specification says should happen. Where Claude differed from the specification, the difference is in the text. Where we couldn't check something, the text says so.

Who this book is for

You're a web developer. You know TypeScript, the DOM, and how to ship a web page. You've probably never written an MCP server, and you don't need to have: Chapter 2 teaches "just enough MCP" and stops as soon as you can build a card.

This isn't a book about MCP in general, and it doesn't survey every host that runs MCP Apps. It's about Claude, on purpose. Claude publishes its own developer documentation for MCP Apps, covering design rules, theming, what happens when a card appears twice, the web address each connector is served from, a size threshold, and directory review. None of that is in the generic MCP documentation. Aiming at one host means every claim can be checked in one place. Appendix C says what carries over to other hosts and what doesn't.

What you need

- **Node.js 20 or later** and npm. The examples use `@modelcontextprotocol/ext-apps 2.0.1` with the 2.x MCP SDK packages; the versions are pinned in the code.
- **A Claude account** that can add a custom connector. We wrote the book with a Pro account on `claude.ai`, in Claude Desktop, and on a phone. Check your own plan's Connectors page to see whether you can add one.
- **A way to give Claude a public HTTPS address for a server on your laptop.** We used a free Cloudflare quick tunnel. Claude reaches connectors from Anthropic's servers, so `localhost`

isn't enough. A tunnel is public, and its address changes every time it restarts, which the book flags where it matters.

- **A browser** for the reference host.

Nothing here costs money beyond a Claude plan you may already have. Publishing to Claude's directory needs a Team or Enterprise organization (Chapter 14), and you can build everything else without one.

Where the code is

Every code example in this book is an excerpt of a real file in the companion code. A script checks each excerpt against its file, byte for byte, before every build, so what you read is what runs.

Download the code: <https://drive.google.com/file/d/1m4XKQfxpWcyo8ExmMCx-bzY98MsulTpV/view?usp=sharing>

The download holds the finished Snug in `code/snug` and, for each chapter, a frozen copy of Snug as it stood at that point in `code/history/chapter-NN`. That means you can start from any chapter, or jump back if you get lost.

Using the code examples

The companion code is released under the MIT License, so you can use it in your own programs and documentation, including commercial ones, without asking. You don't need permission to use a few lines of it, or to quote a short passage of the book with the source named. If you want to reproduce a substantial part of the book's text, or republish the book or its figures, write to the author first: the text and figures are covered by the copyright page, not by the MIT License. Attribution is always appreciated, and looks like this: *MCP Apps for Claude*, by Sandeep Kumar Patel (first edition, 2026).

Getting in touch

Found a stale fact, a bug in an example, or something Claude no longer does the way the book says? The errata are at **tutorialsavvy.com**. When you report a problem, tell us which example or section it's in. For a claim about Claude's behavior, add when you saw it and where.

How the book is organized

Part I, Getting Started, says what an MCP App is in Claude, gives you just enough MCP, and gets Snug's first card running in real Claude.

- **Chapter 1, What Are MCP Apps?**, defines the terms, says where MCP Apps came from and what Claude supports, and shows what you are going to build.
- **Chapter 2, MCP Basics**, builds Snug's text-only server and runs it over HTTP.
- **Chapter 3, Building Your First App**, adds a card, and runs it in basic-host, claude.ai and Claude Desktop.

Part II, Design and Layout, is about the container and the look.

- **Chapter 4, Layouts: Inline, Carousel, and Fullscreen**, builds Claude's three layouts and asks the host which it offers.
- **Chapter 5, Theming Your App**, makes the card take Claude's colors, type, and borders, in light and dark, and checks contrast.
- **Chapter 6, Designing for Mobile**, covers touch, safe areas, and a real Android phone.

Part III, How Your App Behaves, is about what a card does over its life.

- **Chapter 7, The App Lifecycle**, puts every lifecycle event in order and shows which ones Claude sends.
- **Chapter 8, Talking to Claude**, covers pushing context, proposing a message, app-only tools and opening links.
- **Chapter 9, Handling Multiple Copies of a Card**, elects the newest card and silences the rest.
- **Chapter 10, Large and Live Data**, stays under Claude's size threshold, pages results and polls without waste.

Part IV, Security, is about the sandbox and the user.

- **Chapter 11, Sandboxing and Content Security**, declares what the card may load, and audits the bundle to prove it asks for nothing else.
- **Chapter 12, Authentication and Security**, adds sign-in, per-person data, and a threat model.

Part V, Testing and Publishing, is about tests and the directory.

- **Chapter 13, Testing and Debugging**, runs the card in a test with no browser, and shows how to tell whether a test can fail.

- **Chapter 14, Publishing to the Directory**, goes through Claude's review checklist, and prepares Snug's listing.

The appendices hold a message reference (A), a checklist mapping Claude's rules to Snug's code (B), what carries beyond Claude (C), the exercise solutions (D), a troubleshooting guide that starts from what you see on screen (E), and the whole book at a glance (F).

How to read it

Read Part I in order. After that, each chapter tells you what it needs, and Chapters 9 to 12 can be read on their own once you've built the card. Do the exercises. Each has a solution in Appendix D that runs and has a test, and each test was made to fail before it was trusted.

Everything in this book was tried in `claude.ai`, Claude Desktop, and the Claude Android app. Where a chapter passes on advice from Claude's own documentation instead, such as its design guidance, it says so. Claude changes often, so if something you see differs from the book, trust what you see, and check the errata at **tutorialsavvy.com**.

Acknowledgments

This book stands on other people's work, and the parts of it that are checkable are credited here. The MCP Apps extension came out of MCP-UI, created by Ido Salomon and Liad Yosef, and of the OpenAI Apps SDK, as the proposal SEP-1865 records; it is maintained by the MCP project and the contributors to the `ext-apps` repository, whose reference host and SDK the book uses throughout. Anthropic's connector documentation is the source for everything Claude-specific that the book says was documented and not observed.

The companion code runs on Node.js, TypeScript, Vite, Vitest, jsdom, Playwright, zod, Express and jose. The book is set in Literata, Unbounded, and IBM Plex Mono, all published under the SIL Open Font License. The Colophon says how it was built.

Conventions Used in This Book

Code examples are set in a monospace font and numbered by chapter. Example 3-2 is the second example in Chapter 3, with its caption above it. Every example is an excerpt of a real file in the companion code, checked byte for byte against that file before each build, so nothing is pseudocode. An example that is deliberately broken, or that shows output instead of code, says so.

Figures and tables are numbered the same way (Figure 5-1, Table 4-1). Every screenshot is a real capture of Snug running in Claude or in a test host, never a mockup. Where light and dark mode differ, both are shown.

Cross-references use § for a section and "Chapter N" for a chapter, so §8.2 means section 2 of Chapter 8. A script checks every one before each build, so none of them points at nothing.

Version stamps. The code uses `@modelcontextprotocol/ext-apps 2.0.1` and the 2.x MCP SDK packages, and everything in the book was checked against Claude in September 2026. Claude changes often, so if something you see differs from the book, trust what you see, and check the errata at tutorialsvvy.com.

Draft. A feature marked **Draft** comes from the MCP Apps *draft* specification, not the stable one. The text says what Claude did with it when we checked. Draft features can change or disappear.

Note, Tip, and Warning boxes set an aside apart from the text around it. A **Note** adds context you can read now or come back to. A **Tip** is a shortcut or habit that saves time. A **Warning** marks something that can go wrong, or a claim that was true when we checked but is likely to change. Skip a Note on a first read if you like. Don't skip a Warning.

Exercises close every chapter. Their solutions are in Appendix D, and they run: each one is a file in the companion code, with a test.

Snug is the one app the book builds, a meeting-room booking connector. It appears in Chapter 1, first runs in Claude in Chapter 3, and grows with every chapter after that.

PART I

Getting Started

What an MCP App is in Claude, just enough MCP to build one, and Snug's first card running in real Claude.

-
- 01 What Are MCP Apps?
 - 02 MCP Basics
 - 03 Building Your First App
-

Chapter 1: What Are MCP Apps?

Ask a colleague whether a meeting room is free, and you get an answer you can act on: "Orchid's free from half past ten. Want me to book it?" Ask an AI assistant the same question and, until recently, you got a sentence. To book the room you had to type another request, wait, and hope the slot was still there.

MCP Apps change that. Your server can answer with a small interface, right in the conversation, with buttons the person can press. This chapter is the tour before the build: what these apps are, when they're worth the effort, where they run, and the one idea that explains almost every surprise you'll hit later. There's no code yet.

What you'll learn

- What an MCP App is, and why Claude calls it an interactive connector.
- Which problems deserve an interface in the chat, and which don't.
- Where your app will run.
- The big idea of the book: your server ships the interface, and Claude stays in control.

1.1 From Text to Interfaces

Text is a wonderful way to answer a question and a terrible way to act on the answer. The trouble starts the moment a person wants to do something with what they just read.

Picture a tool that queries a database and returns a few hundred rows. Claude can summarize them, but the person usually wants to poke around: sort a column, filter to last week, open one record. With text alone, every poke is another prompt and another round trip. "Show me just last week." "Now sort by revenue." "No, the other column." By the fifth request, you'd rather have a spreadsheet.

Snug, the meeting-room app you'll build in this book, has a smaller version of the same problem. Its first tool, `check_room`, can say "Orchid, next free slot 10:30 to 11:00." The answer is correct, and it's a dead end. To book the room, the person has to type another request, and by the time Claude answers, someone else may have taken the slot. What they want is a **Book 10:30** button right next to the answer, and a quick way to ask for a different time. Building that button is what Part I is about.

Figure 1-1 shows where you're heading: Snug's answer in a real Claude conversation, as a card with the facts and two buttons. It's a real screenshot, and you'll build this exact card.

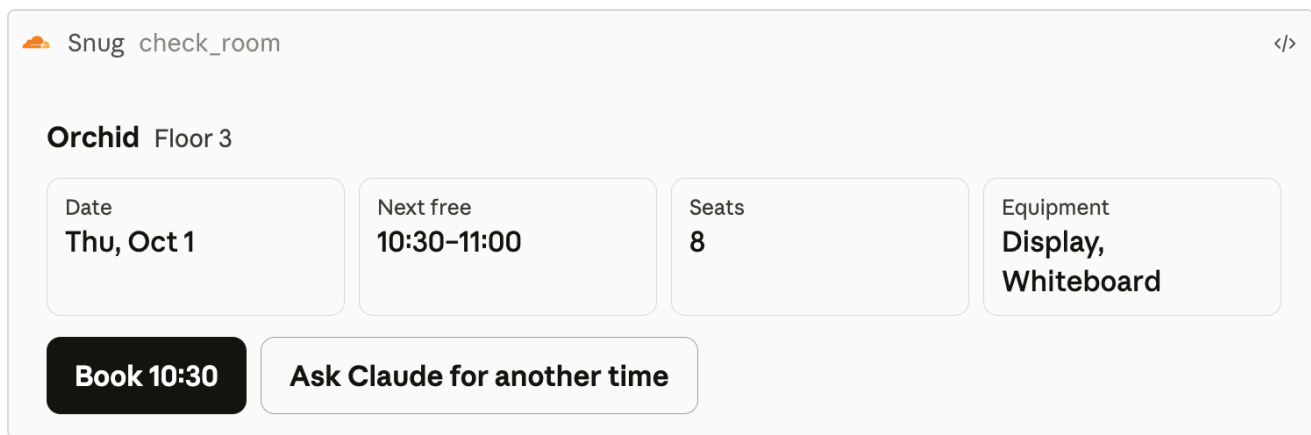


Figure 1-1. What you'll build: Snug's room card, inside a Claude conversation.

Three players make that card appear: Claude, your MCP server, and the card itself. Figure 1-2 shows who talks to whom, and it's the one picture in this chapter worth remembering.

Follow the numbers. The person asks about a room, and Claude decides to call your `check_room` tool (1). Your server answers with a sentence of text and the same facts as data, and it also says which page should display them (2). Claude fetches that page, which is an ordinary HTML file (3), and shows it as a card in the conversation, inside a *sandbox*, a frame that keeps your code apart from Claude's. It passes the card the result (4). From then on the card can call your tools too, for example when the person presses **Book** (5). Those calls always go through Claude, never straight to your server.

That's the full round trip. The rest of the book fills in each arrow, and most of the surprises live in the details of arrows 4 and 5.

A handful of words come up again and again, so here they are in one place for the moments one slips your mind:

- **Host:** the app that runs the conversation and shows your card. In this book, that's Claude: on the web, in Claude Desktop, or in the Claude Android app.
- **Connector:** Claude's name for an MCP server that someone has added to Claude.
- **Tool:** a function on your server that Claude can call, like `check_room`.
- **Resource:** something your server offers for reading. An app's page is a resource, with an address that starts with `ui://`.
- **Card (or view):** your page, as shown in the conversation.
- **Sandbox:** the protected frame the card runs in, kept apart from Claude's own page.

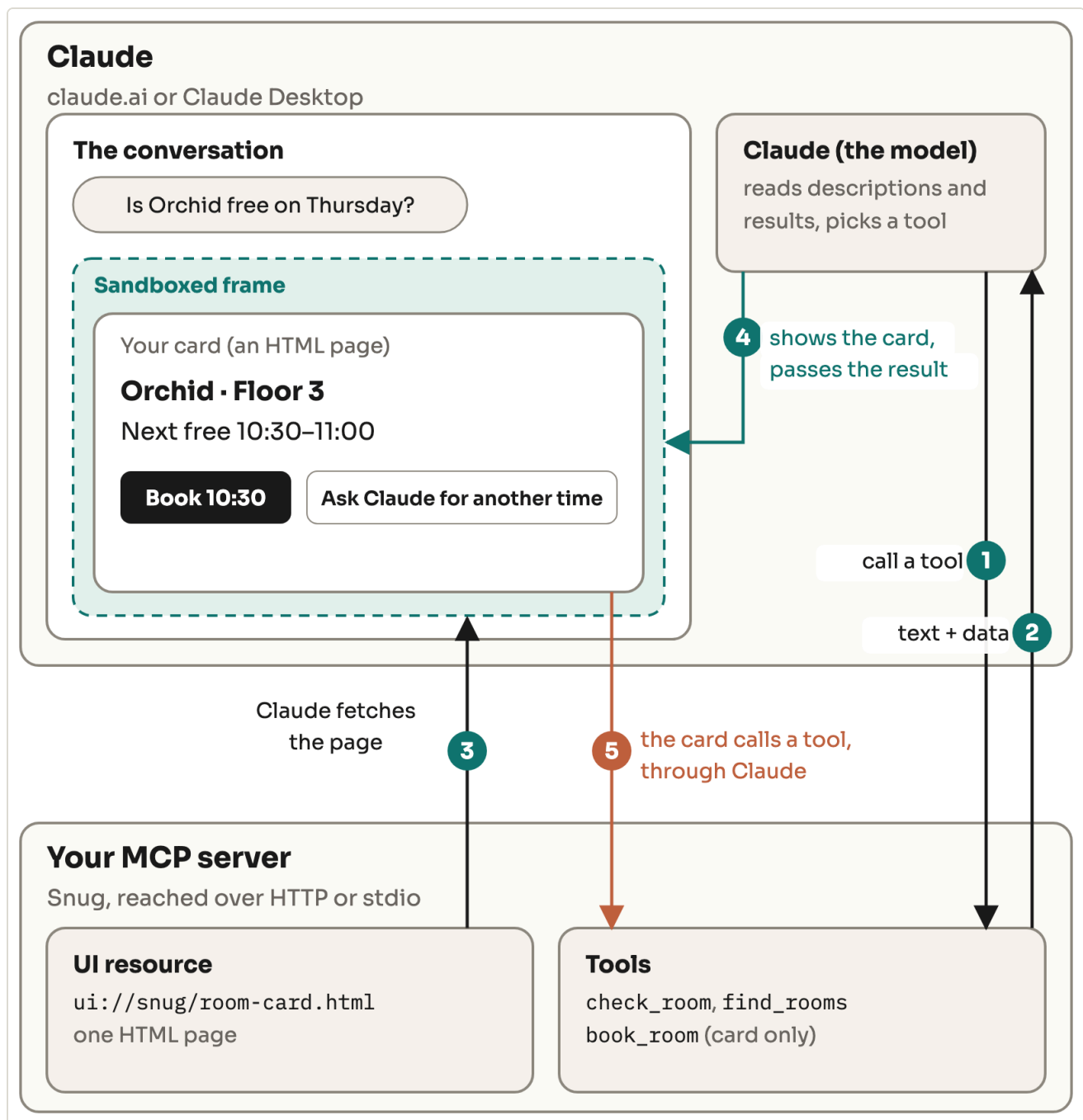


Figure 1-2. How an MCP App works. Claude calls your tool, fetches your page, and shows it as a card in a sandboxed frame. The card can then call your tools too, always through Claude.

The Glossary at the back of the book has the full list.

1.2 One Feature, Two Names

Search for this feature and you'll find two names, and a few forum threads asking whether they're the same thing. They are.

Claude calls it an **interactive connector**. In the Claude Help Center you'll read about connectors that show live, interactive apps inside a conversation, such as dashboards, task boards, and design tools, instead of a text-only reply.

The MCP standard calls it an **MCP App**. It's an official extension to the Model Context Protocol, named `io.modelcontextprotocol/ui`, and the interface itself is an ordinary HTML page, marked with the media type `text/html;profile=mcp-app`. Table 1-1 puts the two names side by side.

The backstory is short. Two earlier projects, MCP-UI and the OpenAI Apps SDK, were solving the same problem in different ways. A proposal called SEP-1865 merged them, and in January 2026 the MCP maintainers announced the result as the first official MCP extension. ChatGPT, Goose, and Visual Studio Code support it too, but this book is about Claude, and the others get one mention, in Appendix C.

Table 1-1. The same feature under its two names.

	CLAUDE	THE MCP STANDARD
Name	Interactive connector	MCP App
Where you read about it	Claude Help Center, Claude developer docs	modelcontextprotocol.io, the <code>ext-apps</code> repository
What you build	A connector (an MCP server) whose tools return a UI	A tool with <code>_meta.ui.resourceUri</code> , and a <code>ui:// resource</code>

1.3 When an App Is Worth It

Not every answer needs an interface, and a card that shouldn't exist is worse than plain text. Our rule of thumb is blunt: build an app when you'd be annoyed to type the follow-up yourself.

The MCP Apps documentation names five kinds of problem where an interface beats text: exploring complex data, configuring something with many options, viewing rich media, watching something that changes, and working through several steps. An app also has advantages over simply linking to a web page. It stays in the conversation, it can call your server without a separate API, and it runs in a sandbox that Claude controls.

Claude's design guidelines are more practical, and they're the ones to design against. Table 1-2 sums them up.

Snug sits comfortably in the third row: booking a room has a clear start and end, and Claude's guidelines use booking as an example. It also involves no money, which matters, because purchases through third-party connectors aren't supported in Claude. That's why this book builds a room booker and not a shop.

Table 1-2. Claude's guidance on what makes a good MCP App.

GOOD FITS	AVOID
Tasks that suit a conversation: data analysis, document review, project coordination	Long or static content that's better read elsewhere
Communication: message search, threads, profiles	Long workflows that don't fit in the space available
Tasks with a clear start and end: booking, ordering, scheduling	Deep navigation: drill-downs, breadcrumbs, many screens
Information the person can act on right away	Scrolling areas nested inside an inline card
Features that genuinely extend what Claude can do	Menus and popovers, which the host may cut off
	A chat box inside your app, which duplicates Claude

1.4 Where Apps Run

Your app runs wherever Claude runs: on the web (claude.ai), in the Claude Desktop app, in the Android app, and in Cowork.

On the web and on desktop, your app appears in a sandboxed frame inside the conversation. On phones it appears in a native web view instead, with no access to the camera, microphone or location, and a connector has to be added on the web or desktop before it shows up on a phone. Table 1-3 shows what this book covers.

Table 1-3. What this book covers.

WHERE CLAUDE RUNS	COVERED IN THIS BOOK
claude.ai on the web	Yes
Claude Desktop	Yes, both as a remote connector and as a local (stdio) server
Claude for Android	Yes: the card, carousel, and booking
Cowork	Not covered

Every screenshot and result in this book comes from `claude.ai`, Claude Desktop, and the Claude Android app.

You may also find two different answers online to the question of *who* can add a custom connector: one guide says you need a paid Claude plan, and another says any account can. This book was written with a Pro account. If you're on another plan, check your own Connectors page.

1.5 Claude Stays in Control

One idea ties the book together, and it saves you the most debugging time. Your card is a guest in Claude's house. You bring the interface, but Claude decides whether it gets in, where it sits, and what it's allowed to touch.

You write the tool, the page, and a few declarations: which domains the page may load from, whether a tool only reads data, whether a tool is private to the app. Claude decides what to do with those declarations, and it makes plenty of decisions you never asked it to make. Table 1-4 lists a few. Each one surprised us at least once while building Snug, and each gets its own section later.

Table 1-4. What you declare, and what Claude decides.

YOU DECLARE	CLAUDE DECIDES
A tool marked <code>readOnlyHint</code>	Whether to ask the person before each call. New connectors start at "Needs approval"
A card that sends a message to Claude	Whether it's sent right away. It waits in the message box for the person to send it
A card that opens a link	Whether to show a confirmation dialog with the full address first
Domains the page may load from	Which of them actually load. Embedded frames, for example, are restricted
An interface to show	Whether to show it, and how tall it can be
An interactive connector	Whether a Team or Enterprise organization allows it at all

Don't fight this. It's what makes it safe for Claude to run code from a server nobody has reviewed, and it's why people can trust a card that shows up in their conversation. It does mean you design for the host: you use its colors, fit its space, expect a permission prompt on the first call, and keep working gracefully when the answer is no. The apps that feel good in Claude are the ones that take the hint.

Note: Claude's behavior can change without notice. The permission defaults in Table 1-4 are a good example. If something in this book doesn't match what you see, trust what you see, and check the errata at tutorialsavvy.com.

1.6 Summary

You now know what an interactive connector is and what to expect from one. In this chapter, you learned the following:

- An interactive connector is Claude's name for an MCP App: a tool whose result is an interface in the conversation, not just text.
- Apps suit tasks with a clear start and end that the person wants to act on. They don't suit long content, deep navigation, nested scrolling, menus, or a chat inside a chat.
- Claude runs apps on the web, on desktop, on phones, and in Cowork. This book builds and tests on the web, on desktop, and on Android.
- You declare; Claude decides. Permissions, messages, links, domains, sizing and organization policy are all Claude's call.

In the next chapter, you'll set up Snug and build its first MCP tool, still without any interface.

1.7 Exercises

Solutions are in Appendix D. These two are written, not coded, because the skill they practice is judgment.

Exercise 1-1. Would a UI help here? For each request, decide whether the best answer is plain text, an inline card, a carousel, a fullscreen view, something an interactive connector can't do, or something that belongs in the chat rather than in a card. Say which guideline decides it. (a) "What time does the office open?" (b) "Is Orchid free at 2?" (c) "Show me rooms for six at 2pm." (d) "Plan next week for Cedar, hour by hour." (e) "Pay the £40 deposit on this booking." (f) "Rewrite my booking note so it sounds friendlier."

Exercise 1-2. Who decided that? Chapter 3 runs Snug in `claude.ai`. List every decision in that walkthrough that Claude made and Snug didn't. Aim for at least five.

Chapter 2: MCP Basics

Snug is going to be a small server that Claude can talk to, and that's all a Claude connector is. This chapter builds the part underneath: one tool that answers a plain question in plain text. By the end, you'll have a running server and you'll have called it yourself, with no Claude involved.

You don't need to learn the whole Model Context Protocol (MCP) to build an app, and this chapter won't try. It covers the handful of ideas an app depends on, and for each one it shows what Claude actually did when it met it. That second part matters. MCP changed a lot in 2026, some tutorials haven't caught up, and in a few places Claude behaves differently from what the official guide promises. When that happens, we'll tell you which one to believe.

What you'll learn

- Write an MCP tool, and understand what it sends back: text for Claude, and data for your app.
- Use tool annotations, and see how Claude uses them to decide when to ask permission.
- Set up a server that works with both the newest and the previous version of MCP.
- Write tool results that still make sense when there's no interface to show them.

2.1 Getting Started with Snug

Before any theory, let's get Snug running on your machine. It takes a few minutes, and from then on you can try everything as you read.

Get the code. Download the companion code from the link in the Preface and unzip it. Inside, `code/snug` holds the finished app, and `code/history` holds a copy of Snug as it stands at the end of each chapter: `chapter-02`, `chapter-03`, and so on. You'll need Node.js 20 or later.

Meet Snug. Snug books meeting rooms. It knows the three rooms in Table 2-1, and each can be booked in half-hour slots between 09:00 and 17:00. A few slots are already taken, so the answers are more interesting than "free all day". Bookings live in memory, which is fine for learning: restart the server and you start fresh.

The booking rules live in `rooms.ts`, and they have nothing to do with MCP: they're ordinary functions such as `findRoom(name)`, `freeSlots(roomId, date)` and `describeAvailability(...)`, which writes a sentence like the one you'll see in a moment. This book doesn't list them, because they'd be the same in any app. The MCP part is the thin layer on top, in `server.ts` and `main.ts`, and that's what this chapter is about.

Table 2-1. Snug's three rooms.

ROOM	FLOOR	SEATS	EQUIPMENT
Orchid	3	8	Display, Whiteboard
Fern	3	4	Display
Cedar	5	14	Display, Video conferencing

Start the server. Open a terminal in the folder where you unzipped the code, and run:

```
cd code/history/chapter-02
npm install
SNUG_TODAY=2026-09-30 npm start
```

The `SNUG_TODAY` setting needs a word of explanation. This book's examples ask about Thursday, 1 October 2026, and Snug won't offer a slot that has already passed. If you're reading after that date, Snug would answer "fully booked", and nothing would match the book. Setting `SNUG_TODAY` makes Snug pretend it's an earlier day, so every example gives the answer printed here. On Windows PowerShell, use `$env:SNUG_TODAY="2026-09-30"` on its own line first. Leave it out when you build your own app.

You should see `Snug (text only)` on `http://localhost:3001/mcp`. Snug is now an MCP server, waiting for requests. The SDK may also print a warning that the server is binding to `0.0.0.0` without DNS rebinding protection. That's expected while you're learning on your own machine. Before you deploy, restrict which hosts the server accepts, with the `allowedHosts` option the warning mentions.

Try it. You don't need Claude to talk to it. In a second terminal, send it the same kind of request Claude would send, asking the `check_room` tool about Orchid:

```
curl -s -X POST http://localhost:3001/mcp \
  -H 'content-type: application/json' \
  -H 'accept: application/json, text/event-stream' \
  -d '{"jsonrpc":"2.0","id":1,"method":"tools/call","params":
    {"name":"check_room","arguments":{"room":"Orchid","date":"2026-10-01"}}}'
```

Snug answers with something like this (shortened, and spread over several lines to fit the page):

```
event: message
data: {"result":{"content":[{"type":"text","text":"Orchid (floor 3, seats 8; Display, Whiteboard) on 2026-10-01: next free slot 10:30-11:00, 12 free half-hour slots left."}],
"structuredContent":{"room":{"id":"orchid","name":"Orchid",...},"date":"2026-10-01","nextSlot":{"start":"10:30","end":"11:00"},"freeCount":12}
}, "jsonrpc":"2.0","id":1}
```

That wall of JSON is the whole conversation between Claude and your server, and you just had it yourself. Two things matter. The request names a tool and hands it some arguments. The answer carries a sentence (`content`) and the same facts as data (`structuredContent`). The rest of this chapter explains both.

Now try to break it. Ask about a room that doesn't exist, such as `"Lotus"`, and see what comes back. You'll meet that answer again in a few pages.

2.2 Tools and Their Results

Somebody asks Claude whether the Orchid room is free on Thursday. Claude has no idea. It doesn't know your rooms, your calendar, or your office, and it needs something that does. That something is a tool: a named function on your server that Claude can call, plus a description that tells Claude when to reach for it.

An MCP server can offer three kinds of things: tools that Claude calls, resources that Claude reads, and prompts. An app needs the first two. Tools come first, and resources come into play once the app has a page, because the page is itself a resource.

What a tool sends back is where the design gets interesting. A result carries `content`, a list of blocks such as text, and `structuredContent`, the same facts as data. If something goes wrong, the result sets `isError: true` and carries a message instead. Figure 2-1 shows the whole round trip for Snug's first tool.

Snug's first tool is `check_room`. Give it a room and a date, and it tells you when the room is next free. Example 2-1 shows the whole of `server.ts`: a function, `createServer`, that builds an `McpServer` and registers the tool on it.

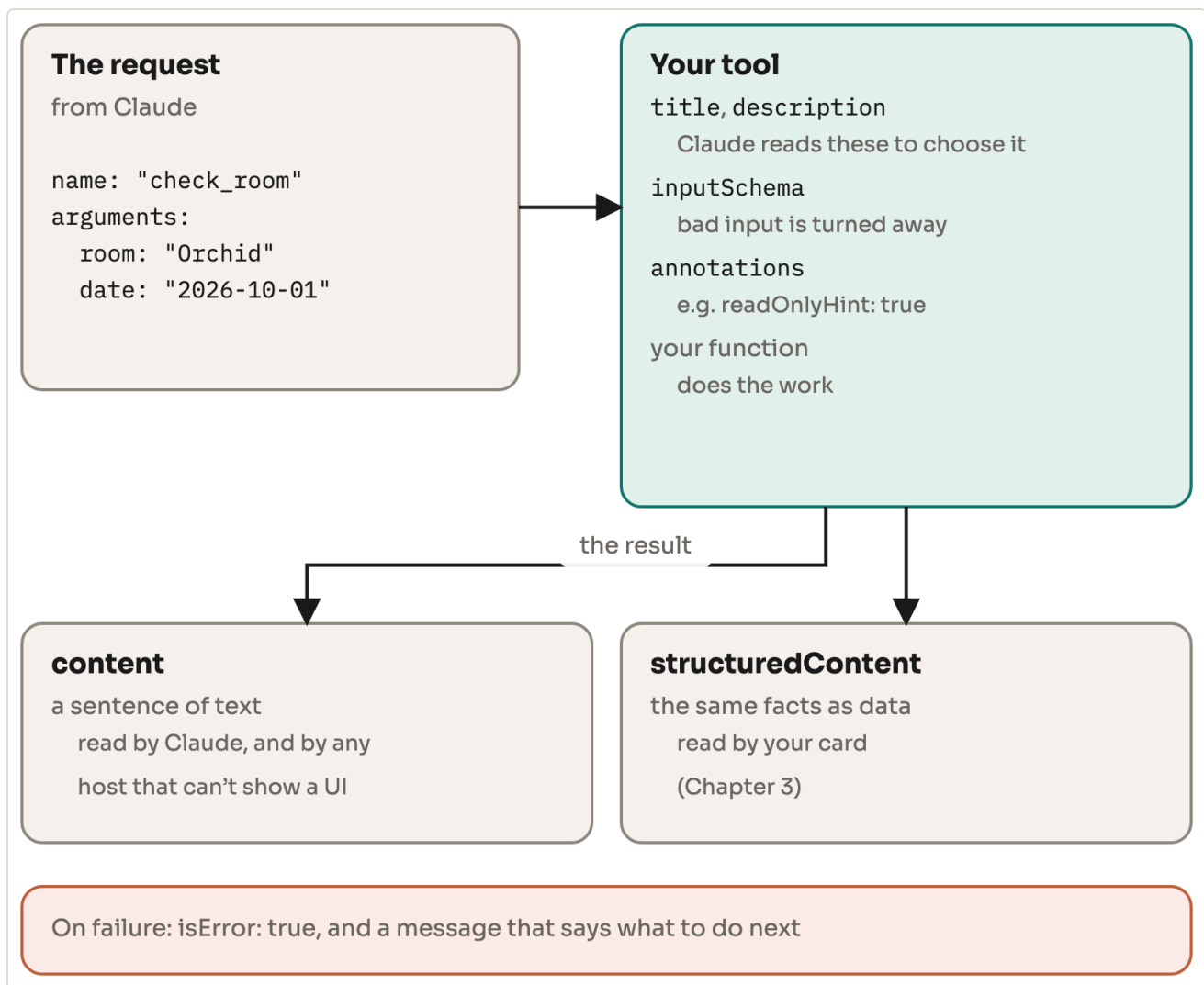


Figure 2-1. A tool call. Claude sends a name and arguments; your tool checks them and returns text for Claude and data for your app.

Example 2-1. `server.ts`: a server with one text-only tool, `check_room`.

```
import { McpServer } from "@modelcontextprotocol/server";
import { z } from "zod";
import { describeAvailability, findRoom, freeSlots, listRooms, today } from "./rooms.js";

const roomNames = () => listRooms().map((r) => r.name).join(", ");

export function createServer(): McpServer {
  const server = new McpServer({ name: "Snug", version: "0.0.1" });

  server.registerTool(
    "check_room",
    {
      title: "Check room availability",
```

```

    description: `A meeting room's next free half-hour slot on a date. Rooms:
    ${roomNames()}.`,
    inputSchema: z.object({
      room: z.string().describe("Room name, e.g. Orchid"),
      date: z.string().regex(/^d{4}-d{2}-d{2}$/, "Use YYYY-MM-DD").optional().describe("Date in YYYY-MM-DD; defaults to today"),
    }),
    annotations: { readOnlyHint: true },
  },
  async ({ room: query, date = today() }) => {
    const room = findRoom(query);
    if (!room) {
      return { isError: true, content: [{ type: "text", text: `No room called "${query}". Rooms: ${roomNames()}.` }] };
    }
    const free = freeSlots(room.id, date);
    return {
      content: [{ type: "text", text: describeAvailability(room, date, free[0], free.length) }],
      structuredContent: { room, date, nextSlot: free[0] ?? null, freeCount: free.length },
    };
  },
);

return server;
}

```

`registerTool` takes three things: the tool's name, a description of the tool, and the function that runs when Claude calls it. Three choices in the description and schema belong in every tool you write.

Start with the description, because it's the only thing Claude reads when it decides whether to call your tool. Snug's lists the valid rooms, so Claude never has to guess a name. Next, the input is a schema, written in zod, which the SDK understands. A badly formed date is turned away before your function runs, so the function never has to defend itself against nonsense. And look at the error branch. When the room doesn't exist, the message doesn't say "not found"; it lists the rooms that do exist, which is what the caller needs next. That isn't just courtesy. Claude's directory review rejects vague errors like "Bad Request".

When the tool succeeds, it answers twice: once as a sentence and once as data. Claude reads the sentence, and later your card will read the data. Section 2.6 explains why that duplication earns its keep.

2.3 Annotations and Permissions

A tool can carry small labels called *annotations*, and two of them matter more than you'd expect, because they decide whether Claude stops to ask the person before running your tool. Nobody likes a prompt they didn't need, and nobody likes a booking they didn't approve. Getting these two right is how you avoid both.

`readOnlyHint: true` says the tool doesn't change anything. `destructiveHint` says whether a tool that does change things could cause damage. Claude's submission checklist asks for a `title` and the right hint on every tool. By the checklist, read-only tools *can* run without asking each time, and destructive tools always ask. It also rejects a single tool that mixes safe and risky actions, so reading and writing belong in separate tools, and tool names stop at 64 characters.

Reality is more cautious than the checklist. `check_room` is marked read-only, and every new Snug connector we added still started it at **Needs approval**. The first time, Claude showed Claude *wants to use `check_room` from Snug*, with the request's details and three choices: **Deny**, **Always allow**, and **Allow once**. After we chose Always allow, it stopped asking.

Warning: A hint describes your tool; it doesn't grant permission. Even a read-only tool may need the person's approval the first time. Make sure your app still works well when that happens.

Snug's second tool, `book_room`, is a harder case. It changes data, so it isn't read-only. Is it destructive? In this chapter's code it says `destructiveHint: false`, since adding a booking destroys nothing. But the checklist describes destructive tools as ones that "modify or delete data", and a booking does modify data. Snug switches this to `true` before it's published, and that's safe.

2.4 Connecting over HTTP or stdio

Claude can reach your server in two ways, and the choice shapes how you test it and how you connect it.

The first is **remote**, over HTTP. Your server listens at a web address, and Claude calls it from Anthropic's servers. That's how `claude.ai` connects, and it's why trying Snug in `claude.ai` needs a tunnel: Anthropic's servers can't see your laptop. The second is **local**, over `stdio`. Claude Desktop starts your server as a program on your computer and talks to it through standard input and output.

The newest version of MCP makes HTTP connections *stateless*. There are no protocol-level sessions any more, so if your server needs to remember something between calls, it hands out its

own identifiers as tool arguments. You'll see this in the server later in this chapter: it builds a fresh server object for each request and keeps no session. A plain GET request, which older versions used for sessions, now gets back `405 Method not allowed`.

2.5 Choosing the Right SDK Setup

This is where the official guides and reality part ways, so we'll take it slowly. Your server needs a few packages from the MCP SDK, which was split into several parts this year, and one function that most tutorials haven't mentioned yet. We found it the hard way.

Version 2 of the TypeScript SDK replaced the single `@modelcontextprotocol/sdk` package with several smaller ones: `server`, for `McpServer` and the request handler; `client`, for connecting to servers and for the in-memory connection used in tests; `node`, for Node's HTTP server; and `express`, for a ready-made Express app. The MCP Apps package, `@modelcontextprotocol/ext-apps` 2.x, depends on all of them and needs Node 20 or later. Some official guides still say to install the old package and Node 18.

Installing the new SDK doesn't make your server speak the newest version of MCP, 2026-07-28. We set one up the classic way, exactly as the ext-apps quickstart shows, and it refused that version outright:

```
Unsupported protocol version: 2026-07-28 (supported versions: 2025-11-25, 2025-06-18, 2025-03-26, 2024-11-05, 2024-10-07)
```

The fix is `createMcpHandler`, a separate function in the SDK. Example 2-2 shows how Snug uses it to serve both old and new clients from one place.

Example 2-2. One handler for two protocol eras (`main.ts`).

```
// Serves protocol 2026-07-28 and falls back to stateless 2025-11-25 for older clients.
// Both come from the same factory, so the two eras can't drift apart.
const mcp = toNodeHandler(createMcpHandler(() => createServer()));
// express.json() has already read the stream, so hand over the parsed body.
app.all("/mcp", (req: Request, res: Response) => mcp(req, res, req.body));
```

`createMcpHandler` takes a function that builds a fresh server. It speaks the newest version of MCP and, by default, also serves older clients from the same server code, so the two can't drift apart. One small wrinkle: the SDK wants to read the request body itself, but Express has already read it, so the last argument passes along the body Express parsed.

To see what Claude actually sends, log every request. Example 2-3 adds a one-line log for each one.

Example 2-3. A request log (`main.ts`).

```
// One JSON line per request: what the host actually sends.
app.use("/mcp", (req, _res, next) => {
  const body = req.body ?? {};
  const params = body.params ?? {};
  const meta = params._meta ?? {};
  console.log(
    JSON.stringify({
      at: new Date().toISOString(),
      method: body.method,
      header: req.get("mcp-protocol-version"),
      protocolVersion: params.protocolVersion ??
        meta["io.modelcontextprotocol/protocolVersion"],
      clientInfo: params.clientInfo ?? meta["io.modelcontextprotocol/clientInfo"],
      tool: params.name,
    })
  );
  next();
});
```

Together, Examples 2-2 and 2-3 make up most of `main.ts`; the rest creates the Express app and starts listening. The full file is in `code/history/chapter-02/main.ts`.

With the log running, you can watch Claude connect. It starts with a quick check using the older handshake. Then it asks for the server's details using the newest version of MCP, lists your tools and resources, and, when the person asks a question, fetches your app's page and calls the tool.

So what happens if your server can't speak the newest version? We tried it, expecting an error. Claude shrugged, fell back to 2025-11-25, and carried on. A server that doesn't know the newest version isn't rejected; it just gets the older one. Table 2-2 sums up what we saw.

Table 2-2. Which MCP version Claude used with Snug.

CONNECTION	TRANSPORT	MCP VERSION USED
claude.ai, Snug using <code>createMcpHandler</code>	HTTP	2026-07-28
claude.ai, Snug using the older setup	HTTP	2025-11-25 (after falling back)
Claude Desktop, local server	stdio	2025-11-25

Note: Claude identifies itself with different client names, such as `Anthropic` , `Anthropic/ClaudeAI` , and `claude-ai` , depending on where the request comes from. Don't write code that depends on the name. It can change, and anyone can send any name they like, so it must never decide who is allowed to do what.

Compare the first and last rows of Table 2-2. It's the same server and the same code, speaking two different versions of MCP depending on how it's connected. Everything you build from here on has to work with both.

2.6 Always Return Useful Text

Sooner or later, your tool will run somewhere that can't show your card: a host without MCP Apps support, or Claude reading the result halfway through a bigger task. Nobody will see your interface there. The text is all anyone gets, so it has to carry the whole answer.

Snug's result does. The sentence in `content` names the room and date, the next free slot and how many slots are left. The MCP Apps documentation makes the same recommendation: a server that offers an interface should still return meaningful text. Text is also the easiest thing to test, because you can read it in a terminal. Example 2-4 shows what a host without an interface receives.

Example 2-4. What a host without a UI receives from `check_room`.

```
Orchid (floor 3, seats 8; Display, Whiteboard) on 2026-10-01: next free slot 10:30-11:00,
12 free half-hour slots left.
```

There's a second lesson, and it caught us off guard. When Snug ran as a local server in Claude Desktop, a coding agent called `check_room` and got back only the `structuredContent`: the room, date, slot, and count as JSON, with no sentence at all. Different callers see different halves of your result. Make both halves complete, and make sure they agree.

2.7 Summary

You now have a working MCP server and a clear picture of how Claude talks to it. In this chapter, you learned the following:

- You can run Snug on your own machine and call its tools with `curl` , no Claude needed.
- A tool result carries `content` for Claude and `structuredContent` for programs. An error sets `isError: true` and includes a message that says what to do next.

- `readOnlyHint` and `destructiveHint` describe your tools. Claude may still ask for approval the first time, even for read-only tools.
- Claude connects over HTTP or stdio. The newest MCP version makes HTTP stateless, so your server keeps no session between calls.
- Use `createMcpHandler` to support the newest version of MCP and older clients from the same code.
- Claude uses the newest version when your server supports it, and falls back gracefully when it doesn't.
- Make your text result complete on its own, and expect some callers to read only the structured data.

In the next chapter, you'll give `check_room` a face: a card that appears right in the conversation, with a button that books the room.

2.8 Exercises

Solutions are in Appendix D, and they run: each one is a file in `code/snug/solutions/` with a test.

Exercise 2-1. A `list_rooms` tool. Add a second tool to Example 2-1's server that lists every room with its floor, seats, and equipment, one line per room. It should be read-only and return text only, with no interface. Test it using the SDK's in-memory connection.

Exercise 2-2. Read your own log. Run the server with the logger from Example 2-3, connect your own Claude to it (Chapter 3 shows how), and ask a question. Then write `summarize(lines)`, which takes the log lines and returns the different client names, the MCP versions seen, and how many times each method was called. It must skip any line that isn't JSON, such as a startup message. Which method came first, and which version did your Claude use?

Chapter 3: Building Your First App

Snug can already answer a question. Ask whether Orchid is free, and its `check_room` tool replies with a sentence: the room, the date, and the next free half hour. That sentence works in every host that speaks MCP. But you can't book a sentence. The person still has to ask again, in words, and hope the slot hasn't gone.

In this chapter you'll give that answer a face. You'll link the tool to a small web page, build a card that shows the result, and add two buttons: one that books the slot and one that hands the conversation back to Claude. Then you'll run the card in three places: a test host on your own machine, `claude.ai`, and Claude Desktop. When you press **Book** in a real Claude conversation and the card changes, you'll know it works.

What you'll learn

- Link a tool to a `ui://` resource, and show a card built from the tool's result.
- Use the `App` class to receive results, call your server, and talk back to Claude.
- Run your app in a local test host, in `claude.ai` and in Claude Desktop.
- Avoid four common mistakes, including one we made ourselves.

The code for this chapter is in `code/history/chapter-03`. To run it, open a terminal in that folder and run `npm install`, then `SNUG_TODAY=2026-09-30 npm start`.

3.1 How an App Fits Together

An MCP App isn't a new kind of server, and that's good news. It's three ordinary pieces that point at each other, and once you can name them, the rest of the chapter is just filling them in.

The **tool** is a normal MCP tool with one extra field, `_meta.ui.resourceUri`, which names a UI resource. The **resource** is a normal MCP resource whose address starts with `ui://` and whose type is `text/html;profile=mcp-app`; its content is a complete HTML page. The **view** is that page once Claude shows it in a sandboxed frame. It receives the tool's result and can call back to your server. Table 3-1 shows where each piece lives in Snug.

The link runs one way: the tool names the resource. That lets Claude fetch the page *before* the tool even runs, and we watched it happen in Snug's log. Claude read the page a fraction of a second before calling the tool, and when the tool ran again in the same conversation, it didn't fetch the page a second time.

Table 3-1. The three pieces of Snug's first card, and where each one lives.

PIECE	WHAT IT IS	FILE
Tool	<code>check_room</code> , which declares <code>_meta.ui.resourceUri</code>	<code>server.ts</code>
Resource	<code>ui://snug/room-card.html</code> , the bundled page	<code>server.ts</code> , serving <code>dist/mcp-app.html</code>
View	The card: the HTML page plus the script that talks to Claude	<code>mcp-app.html</code> , <code>src/mcp-app.ts</code> , <code>src/card.css</code>

The text answer doesn't go away. `check_room` still returns its sentence in `content` and the same facts as data in `structuredContent`. The sentence is for Claude, and for any host that can't show apps. The data is for the card.

Figure 3-1 puts it all in order, from the person's question to a booked room. Bookmark this page. Every example in the rest of the chapter implements one of its arrows, and when something breaks, the figure tells you where to look.

Steps 2 and 3 happen on the server, and you'll write them next. Step 5 is where your card receives the result, in `ontoolresult`. Step 7 is `callServerTool`, where the card asks Claude to call your server for it. And step 8 is `updateModelContext`, where the card tells Claude what happened, so Claude knows about the booking the next time the person asks.

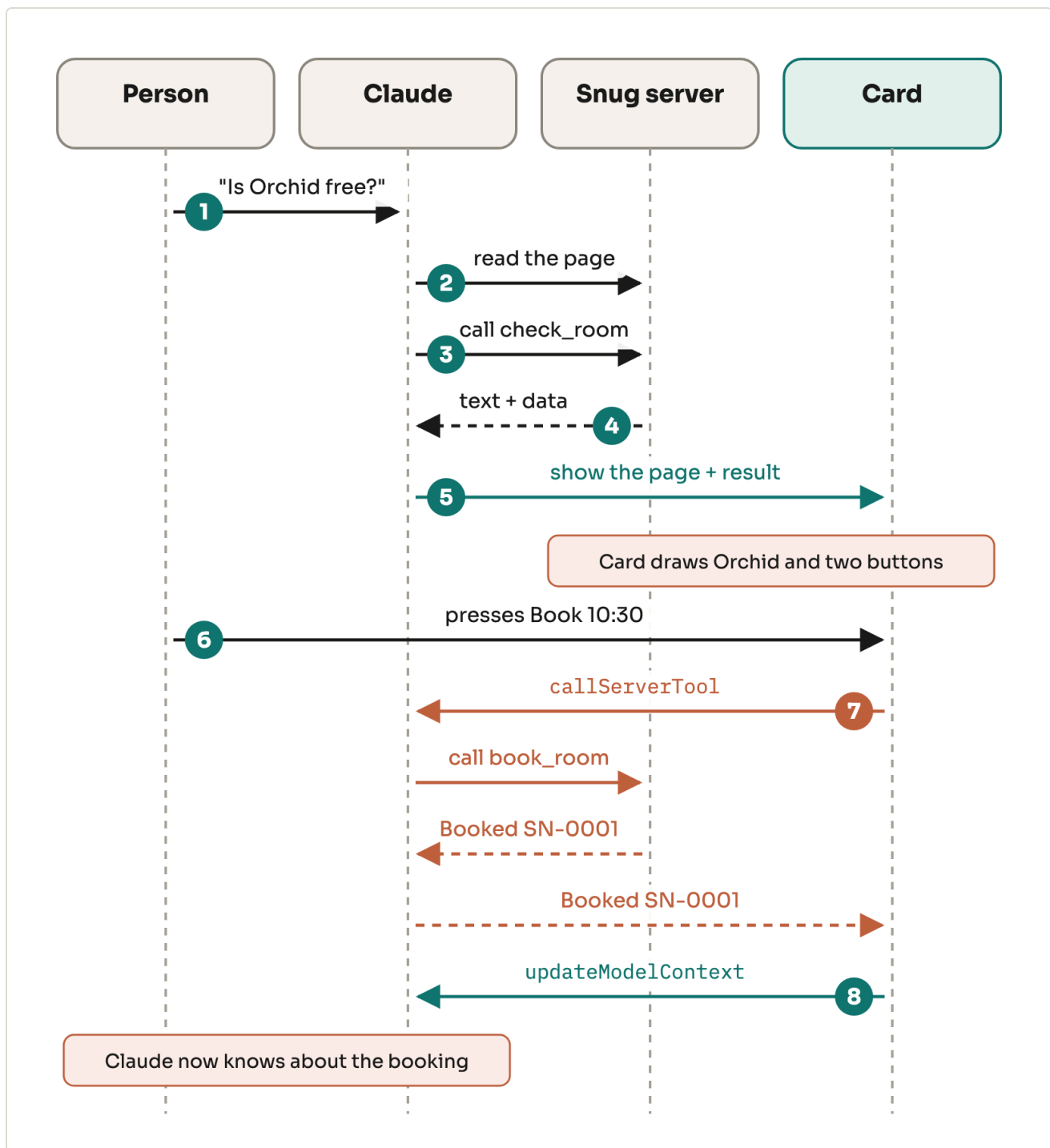


Figure 3-1. From a question to a booking. Claude fetches the page and calls the tool, then shows the card; the card books through Claude and leaves Claude a note.

3.2 The Server Side

The `ext-apps` package gives you two helpers for the server, `registerAppTool` and `registerAppResource`. Both are thin: each registers an ordinary tool or resource and fills in the

app fields for you. Example 3-1 shows the top of the new `server.ts`, where they're imported.

Example 3-1. The top of `server.ts`: imports, two constants, and the MCP Apps declaration.

```
import {
  EXTENSION_ID,
  registerAppResource,
  registerAppTool,
  RESOURCE_MIME_TYPE,
} from "@modelcontextprotocol/ext-apps/server";
import { McpServer } from "@modelcontextprotocol/server";
import fs from "node:fs/promises";
import path from "node:path";
import { fileURLToPath } from "node:url";
import { z } from "zod";
import { book, describeAvailability, findRoom, freeSlots, listRooms, today } from
  "./rooms.js";

const DIST_DIR = path.join(path.dirname(fileURLToPath(import.meta.url)), "dist");
const CARD_URI = "ui://snug/room-card.html";
const DATE = z.string().regex(/^\d{4}-\d{2}-\d{2}$/, "Use YYYY-MM-DD");

const roomNames = () => listRooms().map((r) => r.name).join(", ");

export function createServer(): McpServer {
  // Advertise MCP Apps support where the extensions spec says servers do (server/discover,
  // initialize).
  const server = new McpServer(
    { name: "Snug", version: "0.1.0" },
    { capabilities: { extensions: { [EXTENSION_ID]: {} } } },
  );
```

Three details are new. `CARD_URI` is the address of the card's page, which the tool and the resource will both refer to. `DATE` is the same date check as before, pulled out so several tools can share it. And the server now says, in its capabilities, that it supports the MCP Apps extension (`EXTENSION_ID`). Hosts look for that declaration when they connect.

Example 3-2 shows `check_room`, now linked to the card.

Example 3-2. `check_room`, a read-only tool linked to the card (`server.ts`).

```
// Read-only and visible to Claude. Claude may still ask the person before the first
// call.
registerAppTool(
  server,
  "check_room",
  {
    title: "Check room availability",
```

```

    description: `Show a meeting room's next free half-hour slot on a date as an
interactive card. Rooms: ${roomNames()}.`,
    inputSchema: z.object({
      room: z.string().describe("Room name, e.g. Orchid"),
      date: DATE.optional().describe("Date in YYYY-MM-DD; defaults to today"),
    }),
    annotations: { readOnlyHint: true },
    _meta: { ui: { resourceUri: CARD_URI } },
  },
  async ({ room: query, date = today() }) => {
    const room = findRoom(query);
    if (!room) {
      return { isError: true, content: [{ type: "text", text: `No room called "${query}".
Rooms: ${roomNames()}.` }] };
    }
    const free = freeSlots(room.id, date);
    return {
      content: [{ type: "text", text: describeAvailability(room, date, free[0],
free.length) }],
      structuredContent: { room, date, nextSlot: free[0] ?? null, freeCount: free.length
    },
  },
  };
};
);

```

Three lines carry the weight. The `_meta.ui.resourceUri` line is the entire link between the tool and the card; delete it and you have a plain text tool again. `readOnlyHint: true` tells Claude the tool changes nothing, and you'll see how Claude treats that when you try it in `claude.ai`. And the result carries both `content` and `structuredContent`, as described above.

The **Book** button needs a second tool, one that Claude itself should never call. The `visibility` setting takes care of that.

Example 3-3. `book_room`, a write tool only the card may call (`server.ts`).

```

// App-only write tool: the card's Book button calls it; Claude never sees it in its tool
list.
registerAppTool(
  server,
  "book_room",
  {
    title: "Book room",
    description: "Book a meeting room for one half-hour slot.",
    inputSchema: z.object({
      roomId: z.string(),
      date: DATE,
      start: z.string().describe("Slot start, HH:MM on the half hour"),
    }),
    annotations: { readOnlyHint: false, destructiveHint: false },
  },
);

```

```

    _meta: { ui: { resourceUri: CARD_URI, visibility: ["app"] } },
  },
  async ({ roomId, date, start }) => {
    const result = book(roomId, date, start, "you");
    if (!result.ok) return { isError: true, content: [{ type: "text", text: result.reason
    }] };
    const { booking } = result;
    return {
      content: [{ type: "text", text: `Booked ${booking.id}: ${roomId} on ${date},
      ${booking.slot.start}-${booking.slot.end}.` }],
      structuredContent: { booking },
    };
  },
);

```

visibility: ["app"] is enforced by Claude, not by your server. Snug still lists `book_room` like any other tool, but Claude hides it from the model and shows it under a separate **App-only tools** heading in the connector settings. Resist the urge to hide the tool on the server as well. The card calls it through Claude, so the server has to keep offering it.

The resource is the page itself, read from Vite's build output. Example 3-4 shows how Snug serves it.

Example 3-4. Serving the card as a `ui:// resource(server.ts)`.

```

registerAppResource(server, "Room card", CARD_URI, { mimeType: RESOURCE_MIME_TYPE },
  async () => {
    const html = await fs.readFile(path.join(DIST_DIR, "mcp-app.html"), "utf-8");
    return {
      contents: [
        {
          uri: CARD_URI,
          mimeType: RESOURCE_MIME_TYPE,
          text: html,
          // Blend with Claude's surface; allow the host font origin so applyHostFonts can
          load it.
          _meta: { ui: { prefersBorder: false, csp: { resourceDomains:
            ["https://assets.claude.ai"] } } },
        },
      ],
    };
  });

```

`prefersBorder: false` and the `assets.claude.ai` entry help the card look at home in Claude. For now, look at what's missing: the security policy lists one outside address and nothing else. The card's own script and styles are built into the page itself, so they need no permission at all.

Vite does that bundling with a single plugin. Example 3-5 shows the whole configuration.

Example 3-5. `vite.config.ts`: bundle the view into one self-contained file.

```
import { defineConfig } from "vite";
import { viteSingleFile } from "vite-plugin-singlefile";

// One self-contained HTML file: the view needs no CSP allowances for its own code.
export default defineConfig({
  plugins: [viteSingleFile()],
  build: {
    outDir: "dist",
    emptyOutDir: false,
    rollupOptions: { input: process.env.INPUT },
  },
});
```

The finished card weighs 242 kB, or 64 kB compressed. That's heavy for two buttons and four facts, and almost all of it is the `ext-apps` client library and zod.

3.3 The Client Side

The view is a web page that talks to Claude through `postMessage`. You never touch that directly: the `App` class from `@modelcontextprotocol/ext-apps` handles the conversation for you, and Snug uses four of its methods.

Start with the HTML. It holds a placeholder shaped like the finished card, so there's something on screen while the data loads. Example 3-6 shows the page.

Example 3-6. `mcp-app.html`: the shell, with a loading skeleton.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <!-- Covers first paint: without it the iframe gets an opaque backdrop in Claude. -->
    <meta name="color-scheme" content="light dark" />
    <title>Snug room card</title>
    <link rel="stylesheet" href="/src/card.css" />
  </head>
  <body>
    <main id="card" class="card" aria-busy="true" aria-live="polite">
      <div class="skeleton" aria-hidden="true">
        <div class="bar w40"></div>
        <div class="bar w80"></div>
        <div class="bar w60"></div>
      </div>
    </main>
  </body>
</html>
```

```

    </main>
    <script type="module" src="/src/mcp-app.ts"></script>
  </body>
</html>

```

The `color-scheme` tag and the placeholder both follow Claude's design guidance. The script does the rest. Example 3-7 shows how it starts: it imports the `App` class, describes the data it expects, and creates one `App` for the card.

Example 3-7. The top of `src/mcp-app.ts`: one `App` per card.

```

import {
  App,
  applyDocumentTheme,
  applyHostFonts,
  applyHostStyleVariables,
  type McpUiHostContext,
} from "@modelcontextprotocol/ext-apps";
import type { Booking, Room, Slot } from "../rooms.ts";

interface Availability {
  room: Room;
  date: string;
  nextSlot: Slot | null;
  freeCount: number;
}

const card = document.getElementById("card")!;
const app = new App({ name: "Snug room card", version: "0.1.0" });
let current: Availability | undefined;
let booked: Booking | undefined;

```

The three `apply...` helpers take Claude's theme, colors, and fonts. `current` holds the room the card is showing, and `booked` holds the booking, once there is one. Next, the script sets up its handlers, and connects last.

Example 3-8. Receiving the tool result, then connecting (`src/mcp-app.ts`).

```

// Register before connect() so the first host context update and tool result are not
// missed.
app.addEventListener("hostcontextchanged", (changed) => applyHostContext(changed));
app.ontoolresult = (result) => {
  const a = result.structuredContent as Availability | undefined;
  if (result.isError || !a?.room) {
    renderError(textOf(result) ?? "No availability data.");
    return;
  }
  booked = undefined;

```

```

    render(a);
  };

  // Not a top-level await: Vite's default build target rejects it.
  app.connect().then(() => {
    const initial = app.getHostContext();
    if (initial) applyHostContext(initial);
  });

```

Tip: Always set up `ontoolresult` and your other handlers *before* you call `connect()`. Claude can send the first result the moment the connection opens, and a handler added afterwards can miss it. The bug is silent and intermittent, which makes it the worst kind.

`ontoolresult` receives what the tool returned. Snug reads `structuredContent` and puts it on the page using `textContent` only, never `innerHTML`, so a room name can never turn into HTML. Example 3-9 shows the small helper that builds the page elements.

Example 3-9. Building nodes from data with `textContent` (`src/mcp-app.ts`).

```

// Build nodes with textContent only: tool data never becomes markup.
function el<K extends keyof HTMLElementTagNameMap>(tag: K, className?: string, text?:
  string) {
  const node = document.createElement(tag);
  if (className) node.className = className;
  if (text !== undefined) node.textContent = text;
  return node;
}

```

The **Book** button calls the app-only tool from Example 3-3, then tells Claude what happened.

Example 3-10. Calling a server tool, then updating the model's context (`src/mcp-app.ts`).

```

async function bookSlot(button: HTMLButtonElement) {
  if (!current?.nextSlot) return;
  const { room, date, nextSlot } = current;
  button.disabled = true;
  button.textContent = "Booking...";
  try {
    const result = await app.callServerTool({
      name: "book_room",
      arguments: { roomId: room.id, date, start: nextSlot.start },
    });
    const booking = (result.structuredContent as { booking?: Booking } |
      undefined)?.booking;
    if (result.isError || !booking) throw new Error(textOf(result) ?? "The booking
      failed.");
    booked = booking;
  }
}

```

```

render(current);
// Push: Claude can read the booking on its next turn without being asked to respond
now.
await app.updateModelContext({
  content: [
    { type: "text", text: `The user booked ${room.name} on ${date},
    ${booking.slot.start}-${booking.slot.end} (${booking.id}) from the Snug card.` },
  ],
});
} catch (e) {
  render(current, e instanceof Error ? e.message : String(e));
}
}

```

`updateModelContext()` doesn't make Claude reply. It leaves a note that Claude reads on its next turn. `sendMessage()` is the other way to talk back: it suggests a new message from the person.

Example 3-11. Asking Claude a follow-up question (`src/mcp-app.ts`).

```

async function askClaude() {
  if (!current) return;
  await app.sendMessage({
    role: "user",
    content: [{ type: "text", text: `Find me another time for ${current.room.name} on
    ${current.date}.` }],
  });
}

```

These two behave very differently in Claude, as you're about to see.

3.4 Testing Locally with basic-host

Before trying Claude, run the card somewhere you fully control. The `ext-apps` repository includes **basic-host**, a small test host that shows your app in a sandboxed frame and displays every message passing between the app and the host. If you've ever wished you could see inside the conversation, this is it.

Build and start Snug from `code/history/chapter-03`. Set `SNUG_TODAY`, or the book's dates will be in the past:

```

npm install
npm run build
SNUG_TODAY=2026-09-30 npm run serve

```

Snug is now listening on `http://localhost:3001/mcp`. Next, get basic-host. In a second terminal, clone the `ext-apps` repository and install it:

```
git clone https://github.com/modelcontextprotocol/ext-apps.git
cd ext-apps
npm install
```

Then start basic-host with Snug's address:

```
cd examples/basic-host
SERVERS='["http://localhost:3001/mcp"]' npm start
```

Open `http://localhost:8080`. Choose `check_room`, enter `{"room": "Orchid", "date": "2026-10-01"}`, and call it. (A fixed date means the card looks the same whatever time of day you try it.) The card appears. Press **Book**, and basic-host's Model Context panel shows the note from Example 3-10. Press **Ask Claude for another time**, and the Messages panel shows the follow-up from Example 3-11. Switch basic-host to dark mode, and the card follows along. Figure 3-2 shows the card after booking.

basic-host proves that your app speaks the protocol correctly. It can't tell you how Claude behaves, and the two don't always agree. Time for the real thing.

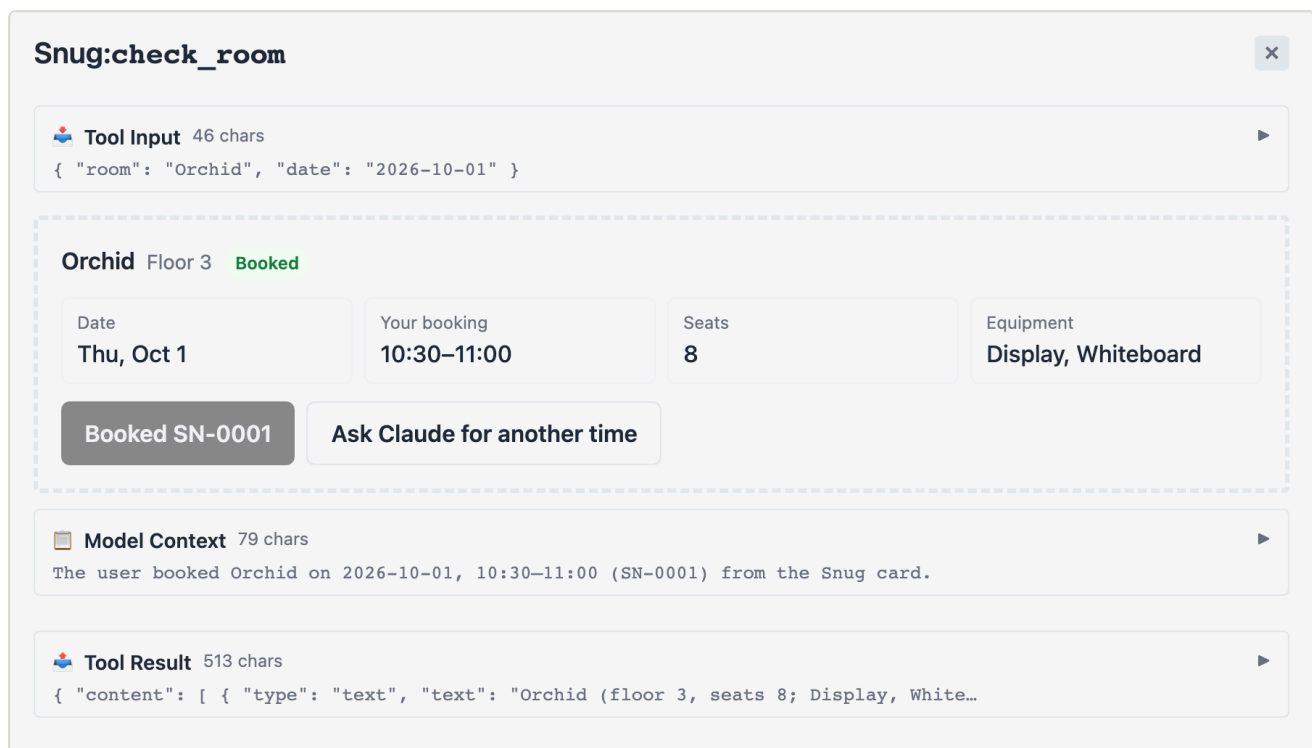


Figure 3-2. Snug's card in basic-host after booking. The Model Context panel shows the note from Example 3-10.

3.5 Running Your App in claude.ai

claude.ai connects to your server from Anthropic's servers, not from your browser, so `localhost` won't do. Snug needs a public HTTPS address, and the quickest way to get one is a Cloudflare tunnel. With Snug running, open a second terminal and run:

```
npx cloudflared tunnel --url http://localhost:3001
```

`cloudflared` prints an address ending in `trycloudflare.com`. Add `/mcp` to the end, and that's your connector URL.

Warning: Anyone who knows the tunnel address can use your server while the tunnel is running. The address also changes every time you restart the tunnel. Close it when you're done, and use a proper deployment before sharing your app.

In claude.ai, open **Customize**, then **Connectors**. Click **+ Add** at the top right and choose **Add custom connector**. Enter a name (Snug) and the URL, then click **Continue**. Claude checks the server and asks how people sign in. For Snug it picks **No sign-in**, with a reminder that anyone with the URL can use it. Click **Add**, then **Connect**.

You can't edit a connector's URL later, only remove it. So each time your tunnel restarts, remove the connector and add it again with the new address. Figure 3-3 shows Snug's connector page.

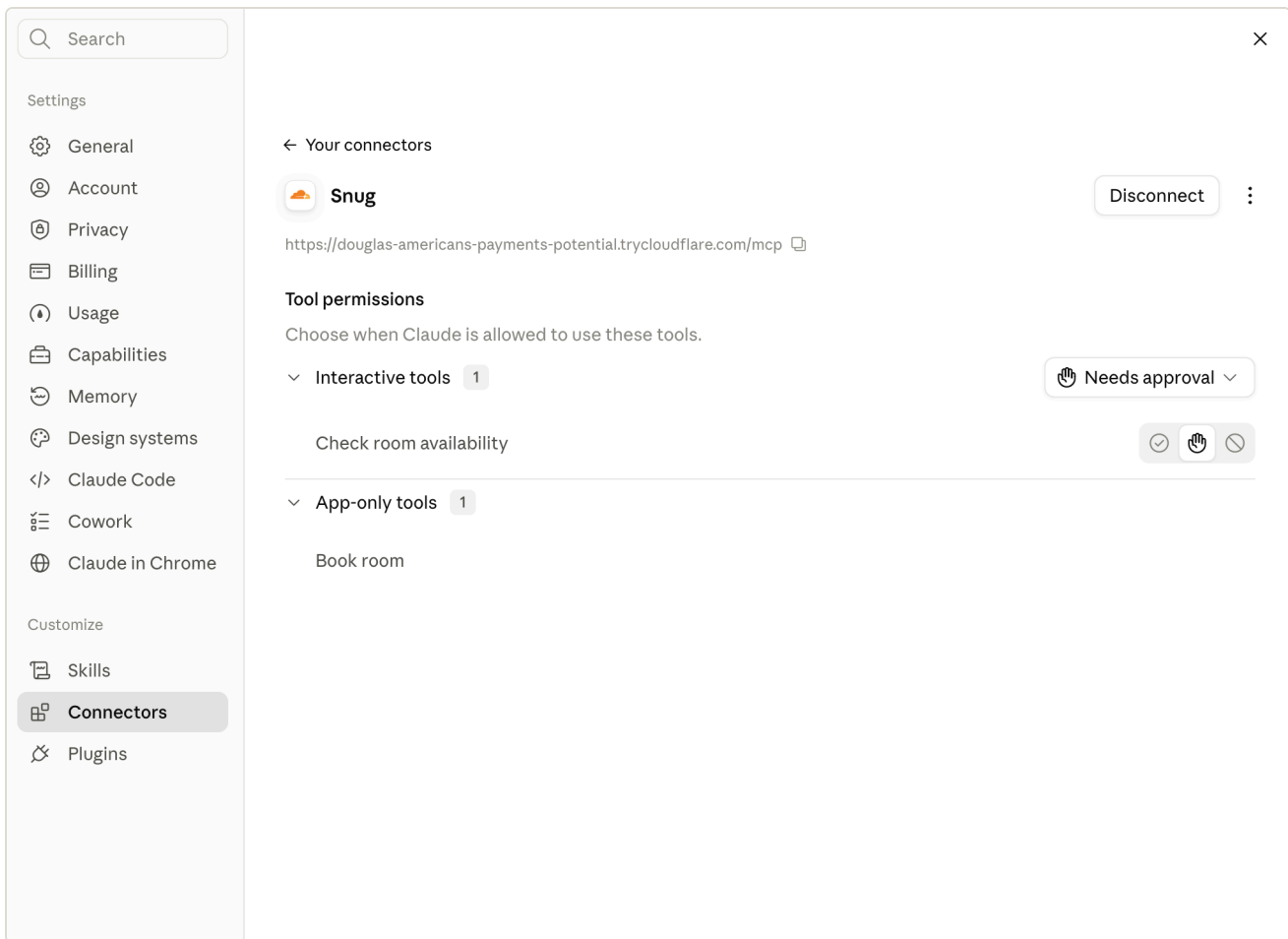


Figure 3-3. Snug's connector page in claude.ai. `check_room` is listed under Interactive tools and `book_room` under App-only tools.

Each interactive tool has a permission setting with three choices: always allow, ask each time (**Needs approval**) or block. New connectors started `check_room` at **Needs approval**, even though it's marked read-only. The person can switch it to **Always allow**, but you shouldn't count on that. Design your card so it works fine when the first call needs approval.

Now start a new chat and ask, "Is the Orchid meeting room free this afternoon?" Sometimes Claude asks first whether it should use Snug; mention Snug by name, and it usually goes straight to the tool. With the permission set to Needs approval, you'll see Claude wants to use `check_room` from Snug, with the details of the request and three buttons: **Deny**, **Always allow**, and **Allow once**. Approve it, and the card appears right in the conversation, with no border and in Claude's own typeface, as Figure 3-4 shows.

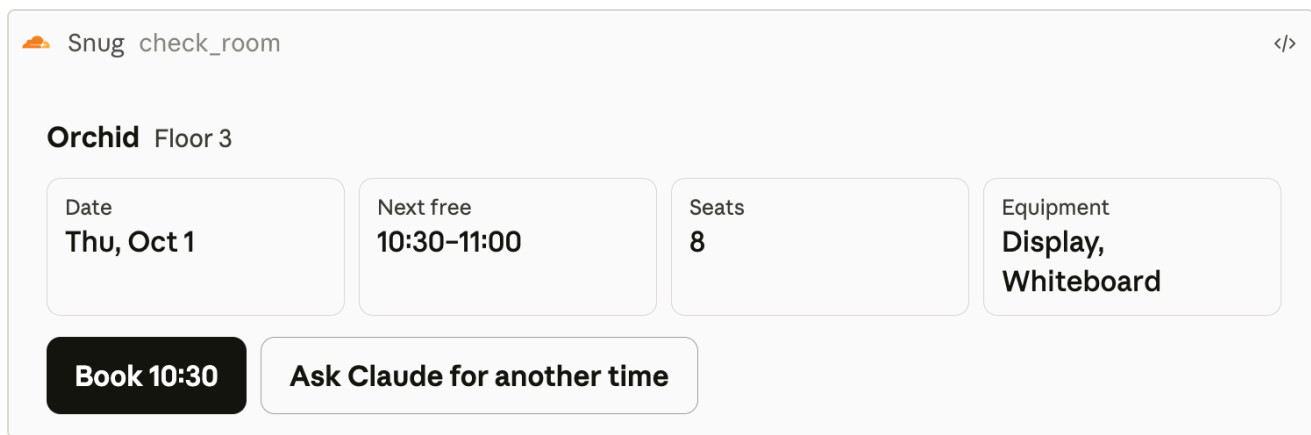


Figure 3-4. Snug's room card in claude.ai. It has no border and uses Claude's own typeface.

Press **Book**. The card changes to Booked SN-0001, and Claude doesn't ask for permission, because `book_room` is an app-only tool. That small moment is the whole point of the book: you pressed a button inside a conversation, and something real happened. (If a click seems to do nothing, check that the browser tab is in front. A card in a background tab may not receive clicks.)

Now press **Ask Claude for another time**, and prepare to be surprised. Nothing is sent, at least not yet. Claude puts the text in the message box, shows a warning that content from apps could try to trick it, and waits for the person to press Send. Once they do, Claude reads the note the card left earlier and replies that the room is already booked for the time you took. Then it calls `check_room` again, and a second card appears below the first.

3.6 Running Your App in Claude Desktop

Claude Desktop can connect to the same tunnel address, as claude.ai does. It can also run your server as a local program, talking over standard input and output (stdio) instead of HTTP. That needs no tunnel, and nothing is exposed to the internet. Example 3-12 shows how Snug picks between the two.

Example 3-12. Snug's entry point chooses stdio or HTTP (`main.ts`).

```
// Local mode for Claude Desktop: stdin/stdout carry the protocol, so keep stdout for
// protocol messages only.
if (process.argv.includes("--stdio")) {
  await createServer().connect(new StdioServerTransport());
} else {
  startHttp(parseInt(process.env.PORT ?? "3001", 10));
}
```

To add a local server, open Claude Desktop's **Settings**, then **Developer**, then **Edit Config**. This opens `claude_desktop_config.json`. Add Snug under `mcpServers`, with the path changed to match your copy:

```
{
  "mcpServers": {
    "snug": {
      "command": "npx",
      "args": ["tsx", "/path/to/code/history/chapter-03/main.ts", "--stdio"]
    }
  }
}
```

Build the card first (`npm run build`), because the resource reads it from `dist/`.

Warning: Quit Claude Desktop before you edit this file. If the app is running, it may save its own copy of the settings over yours, and your new entry will quietly disappear.

After a restart, Claude Desktop starts Snug and reports *Connected to snug* (2 tools). Only one of them is offered to the model: `book_room` stays hidden, just as its `visibility` setting asks. If a local server doesn't show up, look at its log file, `~/Library/Logs/Claude/mcp-server-snug.log`. It records every message in both directions.

Now ask Claude Desktop whether Orchid is free on 2026-10-01. The card appears inline, just as it did in `claude.ai`, and Claude adds a short text answer below it (Figure 3-5). Behind the scenes, Claude Desktop tells the card that it's running on `desktop`, in the light theme, in a space 659 pixels wide.

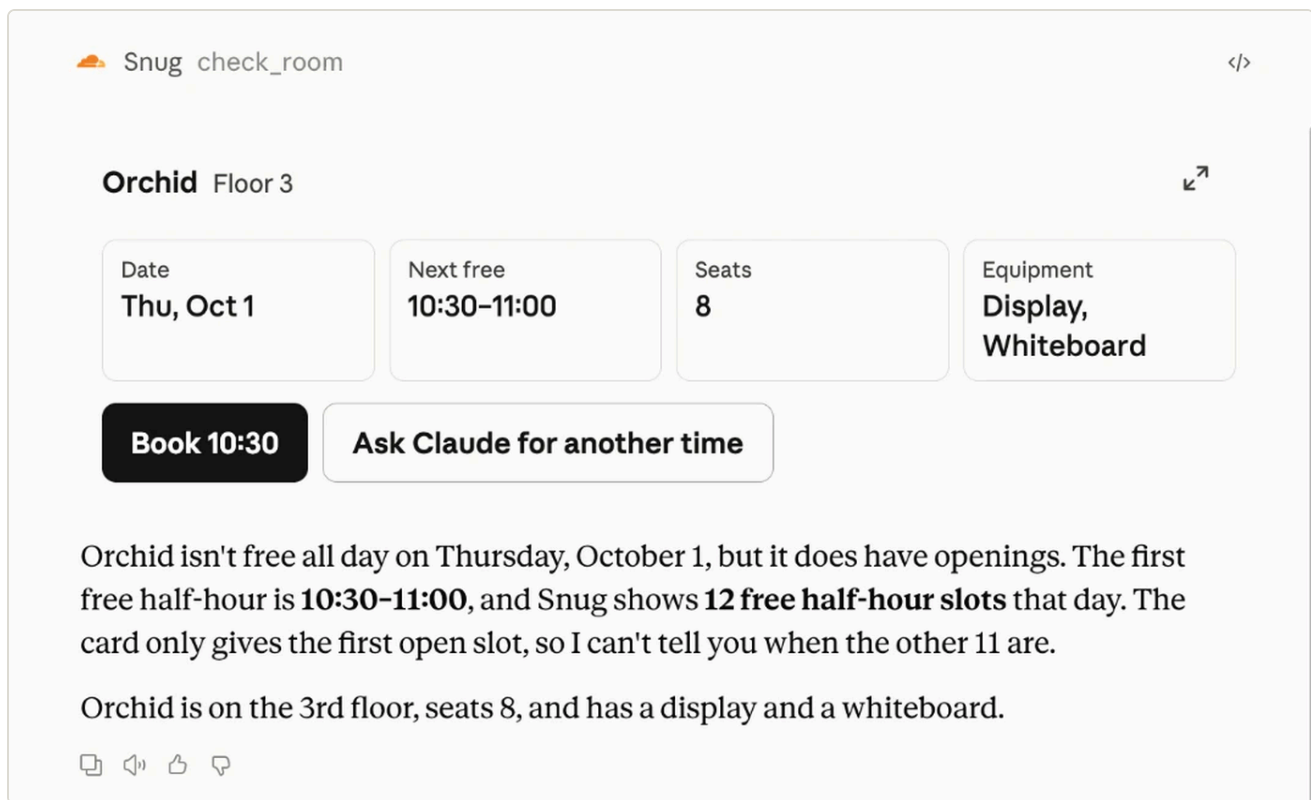


Figure 3-5. The same card in Claude Desktop, with Claude's text answer under it.

If you connected through the tunnel and Claude says the connector "failed to connect" with a 502 error, your tunnel has stopped. Restart it and add the connector again with the new address.

One last thing about stdio. When your server talks over standard output, that channel belongs to the protocol. You'll often read that a single stray `console.log` will break a stdio server. In our tests the SDK's client skipped a stray line without complaint, but don't rely on that. Keep standard output for protocol messages, and send your own logging to standard error.

3.7 Four Common Mistakes

Four things trip people up here. You'll run into the first three in tutorials online. The fourth is a mistake we made ourselves in Snug's first version, and it taught us the most.

Following an out-of-date quickstart. The official *Build an MCP App* guide asks for Node.js 18 and imports from `@modelcontextprotocol/sdk`. The current `ext-apps` (2.0.1) needs Node 20 or later and uses the newer, split SDK packages: `@modelcontextprotocol/server`, `client`, `core`, `node`, and `express`. Snug uses the split packages throughout.

Using `await app.connect()` at the top of a module. Some examples write this. With Vite 6's default build settings, it fails with a build error. Example 3-8 uses `app.connect().then(...)` instead,

which works everywhere.

Assuming you need a paid plan. One guide says custom connectors need a paid Claude plan, and another says any account can add one. We used a Pro account, so check the Connectors page on your own plan.

Forgetting what time it is. Snug's first version of `freeSlots` started every day at 09:00. At 11:54 one morning, it cheerfully offered a 10:30 slot, an hour and a half in the past, and it would have booked it. Every test passed, because none of them looked at the clock. The fix is to make "now" part of availability. Example 3-13 shows it.

Example 3-13. Past slots are never free (`rooms.ts`).

```
return rooms.find((r) => r.id === q || r.name.toLowerCase() === q);
}

/** Earliest bookable minute on a date: the next half-hour boundary if the date is today.
 */
function earliest(date: string): number {
  if (date < today()) return Infinity;
  if (date > today()) return OPEN;
  const now = new Date();
```

Snug's tests now fix the clock at 11:54 and check both sides: no slot before 12:00 is offered, and trying to book 10:30 is refused with the next free time. Put the old `freeSlots` back, and four of those tests fail, which is what you want from a test.

3.8 Summary

Snug's answer now has a card, and the card works in a test host, in `claude.ai` and in Claude Desktop. In this chapter, you learned the following:

- An MCP App has three pieces: a tool that names a `ui://` resource, the resource that holds the page, and the view Claude shows.
- `registerAppTool` and `registerAppResource` fill in the app fields for you. `visibility: ["app"]` keeps a tool away from the model while the card can still call it.
- Set up the `App` class's handlers before calling `connect()`. `callServerTool` reaches your server, `updateModelContext` leaves Claude a note, and `sendMessage` suggests a message.
- `basic-host` shows every message between app and host, so start there.
- In `claude.ai`, new connectors ask for approval before the first call, app-only tools run without a prompt, and suggested messages wait for the person to send them.

- Claude Desktop can use the same remote connector, or run your server locally over stdio.
- Watch for outdated quickstarts, top-level `await`, the assumption that you need a paid plan, and code that forgets the current time.

In the next chapter, you'll go beyond a single card, with a carousel for comparing rooms and a fullscreen view for a whole day.

3.9 Exercises

Solutions are in Appendix D, and they run: each one is a file in `code/snug/solutions/` with a test.

Exercise 3-1. Tomorrow. Late in the day, Orchid's card says *None today* and the Book button is disabled. Write `nextAvailable(roomId, from, days)`, which returns the first free slot on `from` or on any of the following days, up to `days` days, or `undefined` if there is none. Test it with the clock fixed: at 16:50 on 2026-09-25, it should return 10:30 on 2026-09-26.

Exercise 3-2. Break it on purpose. Remove `app.connect()` from the card, and predict what happens when the host sends a tool result. Is there an error, and who sees it? Then check your prediction without a browser: connect an `AppBridge` from `@modelcontextprotocol/ext-apps/app-bridge` (playing the host) to an `App` (playing the card) using the SDK's `InMemoryTransport`, once with `connect()` and once without.

PART II

Design and Layout

Display modes, Claude's tokens and type, and the phone in the user's pocket.

04 Layouts: Inline, Carousel, and Fullscreen

05 Theming Your App

06 Designing for Mobile

FREE SAMPLE

End of the sample

You have read the front matter and the first three chapters of *MCP Apps for Claude*. The full book continues with:

Part II: Design and Layout

Chapter 4: Layouts: Inline, Carousel, and Fullscreen

Chapter 5: Theming Your App

Chapter 6: Designing for Mobile

Part III: How Your App Behaves

Chapter 7: The App Lifecycle

Chapter 8: Talking to Claude

Chapter 9: Handling Multiple Copies of a Card

Chapter 10: Large and Live Data

Part IV: Security

Chapter 11: Sandboxing and Content Security

Chapter 12: Authentication and Security

Part V: Testing and Publishing

Chapter 13: Testing and Debugging

Chapter 14: Publishing to the Directory

Appendix A: MCP Apps Message Reference

Appendix B: Snug Checklist

Appendix C: Beyond Claude

Appendix D: Exercise Solutions

Appendix E: Troubleshooting

Appendix F: MCP Apps at a Glance

Glossary

About the Author

Colophon

Every chapter ends with exercises, and Appendix D has runnable solutions to all of them.