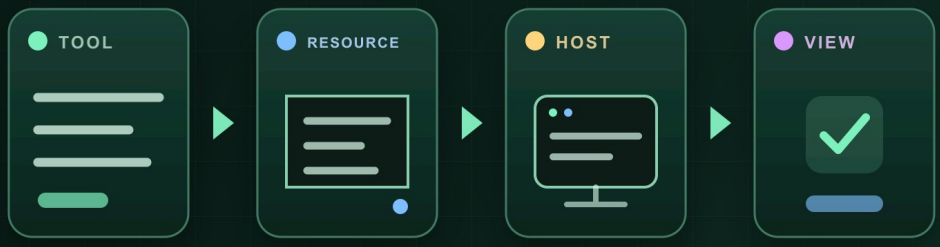
PROJECT GUIDE

BUILD MCP APPS WITH REACT

A Project-Based TypeScript Guide to Interactive Dashboards, Forms, and Tool UIs for AI Chat Hosts



ONE COMPLETE TESTED PROJECT

Structured results • Text fallback • App-only tools
Host context • CSP • Accessibility • Testing

TINTCAIRN STUDIO

FIRST EDITION

Copyright

Software interfaces change. The examples were built against the versions recorded in the companion project and source log. Check current official documentation before shipping a production system.

This book teaches software architecture and implementation. It does not provide legal, security-certification, or compliance advice. The sample data is fictional.

AI tools were used to generate and revise the text and to help create the code-based cover artwork. The manuscript and companion project were extensively edited, built, tested, visually reviewed, and checked against the cited primary sources. Any publishing-platform disclosure about generated content must be answered truthfully.

Contents

Who this book is for

This book is for frontend and full-stack developers who already know the basics of React and TypeScript and understand why an MCP server exposes tools. You do not need to have built an embedded chat interface before. You do need to be comfortable reading a small TypeScript project, running npm scripts, and following data across a browser and a server boundary.

The book is deliberately not a complete introduction to React, TypeScript, Node.js, or the Model Context Protocol. The official documentation serves those goals better and changes faster than a book can. Our narrower job is to bridge the gap between a working tool and a useful interactive surface.

What you will build

The companion project is a Release Readiness Board. A model or user asks the server to review a software release. The server validates a compact set of release checks, calculates a deterministic readiness result, and returns both structured data and a readable text summary. A capable host renders the result as an interactive React dashboard. From that dashboard, the user can record a fictional release decision through a second tool that is visible only to the app.

The project is intentionally local and self-contained. The app makes no external network request, loads no remote font or image, and carries no secret. That constraint creates a clean learning environment and a strong default architecture.

By the end, you will be able to explain and implement:

- the Tool plus UI Resource pattern;
- a text fallback that remains useful without a rendered app;
- stable structured results that are safe to render;
- a React lifecycle connected to tool input and results;
- app-to-server calls for explicit mutations;
- host theme and safe-area adaptation;
- a minimal content security policy;
- tests for domain logic, protocol registration, and host behaviour;
- a release bundle another developer can inspect and reproduce.

How to use the companion

Read the chapters in order on a first pass. Keep the companion repository open beside the book. Each chapter names the files that carry the idea. Run the tests whenever you make a change. Later, use the production checklist as a review guide for your own app.

The exact commands and package versions live in the companion README. If an API has moved since this edition, prefer the official SDK migration notes over forcing an old signature to work.

1 The App Is More Than an Iframe

An MCP App can look simple from the outside. A tool returns a result, a panel appears in the conversation, and the panel contains buttons and charts. It is tempting to describe that as an iframe attached to a tool call.

That description is technically suggestive and architecturally weak. It hides the boundaries that determine whether the result is portable, accessible, secure, and useful when a host cannot render the interface. A better mental model has four participants: the server, the host, the view, and the model. Each participant knows different things and is allowed to do different work.

The four participants

The server owns tools, validation, domain rules, and durable side effects. It receives a tool call through MCP, applies the rules of the application, and returns a tool result. It also exposes the HTML document that implements the interactive view as an MCP resource.

The host is the conversational product embedding the app. It connects to the MCP server, invokes tools, obtains resources, creates a sandboxed frame, and brokers messages between the frame and the wider conversation. The host may supply context such as theme, available display modes, locale, or safe-area insets. Capabilities differ between hosts, so the app must detect what is available instead of assuming one vendor's behaviour.

The view is the code running inside the embedded frame. In our project it is a React application. It receives the original tool input and result through the host bridge, renders useful controls, and can ask the host to call an allowed server tool. The view does not inherit the authority of the host page. It does not get to reach into the conversation DOM, read host cookies, or quietly acquire credentials.

The model helps a person discover and invoke the tool. It can explain the result and continue the conversation. It is not the source of truth for a score, an approval, or a recorded decision. When a business rule must be deterministic, the server computes it. When a person must explicitly confirm a mutation, the interface asks and the server validates.

The flow looks like this:

```
person and model
      |
      v
host and MCP client
      |
      v
server tool  ----  UI resource
      |              |
      +------- result --+
                     v
               sandboxed React view
                     |
                     +-- optional app-only tool call --> server
```

The arrows matter. The server does not push arbitrary page code into the top-level host. The view does not call server functions directly as imported JavaScript. Messages cross explicit protocol boundaries.

Tool plus UI Resource

The core MCP Apps pattern is a tool linked to a UI resource. The tool is registered with metadata containing the resource URI. The resource is registered separately and returns a bundled HTML document with the MCP Apps resource MIME type. When a compatible host calls the tool, it can follow that link, fetch the HTML through MCP, and render the view.

This separation produces two useful properties.

First, the tool remains a tool. A host that does not support Apps can still call it and use its result. An automated client can invoke it without pretending to be a browser. Tests can exercise the domain result without rendering React.

Second, the view is versionable content. The server chooses which resource URI and document it returns. That makes the UI an explicit part of the server contract instead of an accidental external website.

In the Release Readiness Board, the primary tool is named `review_release`. Its input contains a release name and a small list of checks. Its result contains a score, a recommendation, a normalised check list, and a short set of blocking reasons. Its metadata points to `ui://release-readiness/board.html`.

The resource at that URI returns the compiled dashboard. No external script tag is needed. The browser code and CSS are folded into one HTML file during the build.

The text result is part of the product

Do not treat the text content of a tool result as a loading message for the real interface. It is the fallback interface.

A text-only host should still tell the user whether the release is ready, what the score means, which checks block it, and what action to take next. A screen reader may encounter the conversation result before the embedded view. A user may quote the tool response in another conversation. A logging or evaluation system may retain only the content array.

Our result therefore has two coordinated forms:

- `structuredContent` gives the React view stable fields to render;
- `content` contains a concise, human-readable summary with the same conclusion.

The two forms must not contradict each other. Both are generated from the same pure domain result. If the recommendation says hold in structured data, the text cannot say the release is approved.

This is progressive enhancement in a conversational setting. The app makes a strong result easier to inspect and act on; it does not rescue an otherwise useless result.

The example conversation

Imagine a developer asks, “Review release 2.4.0. Unit tests and rollback are ready, but the security review is still pending.” The model selects `review_release` and passes a typed list of checks.

The server normalises the input and computes the result. Because a blocking check is pending, the recommendation is hold. The text response names the blocker. In an Apps-capable host, the board appears with a score, a visible hold state, and the individual checks. The security row is easy to find without reading a paragraph.

After the team resolves the check, the developer can rerun the review. If the result is ready, a button allows the user to record a synthetic decision. Clicking it does not mutate local React state and hope for the best. The view calls the app-only `record_release_decision` tool. The server validates the decision and returns a receipt. Only then does the view show confirmation.

That small interaction demonstrates the complete pattern: discovery through a normal tool, rich inspection through a view, and explicit action through another server tool.

What the example does not do

The companion is not a deployment platform, ticketing system, security scanner, or compliance engine. It stores no real release decision and talks to no vendor API. The mutation tool returns an in-memory demonstration receipt so the protocol path can be tested without credentials or infrastructure.

Those omissions are deliberate. They keep the example focused on the boundary between an MCP tool and an app. They also prevent a tutorial convenience from becoming an unsafe production recommendation.

When you adapt the pattern, you will decide which real system owns each check, how users authenticate, how decisions are authorised, and how idempotency works. The architecture in this book gives you places to make those decisions. It does not make them for you.

A practical definition of done

Before moving on, hold the project to a higher bar than "the panel rendered once." A first useful MCP App should satisfy these statements:

- The primary tool is valuable in a text-only client.
- The linked resource resolves to one self-contained document.
- The view handles connecting, result, error, cancellation, and empty states.
- Business calculations are deterministic and tested outside React.
- A server-side mutation is explicit and validated.
- The interface works with a keyboard and exposes meaningful status text.
- No secret or unnecessary external origin exists in the view.
- Another developer can build and inspect the project from documented commands.

That is the product we will build through the rest of the book.

2 Draw the Boundary Before the Interface

The quickest way to make an embedded app fragile is to start with components. You can produce an attractive card grid before deciding which process owns the score, which input is trusted, or what a button is authorised to change. Those decisions then leak into event handlers and become expensive to repair.

Start with boundaries. A one-page design note is enough for this project. It should answer four questions:

1. What data enters the system and from whom
2. Which component is authoritative for each derived value
3. Which operations are reads and which are mutations
4. What must still work when the rich view is absent

Map authority not just data flow

The release name and check statuses arrive as tool input. They may be proposed by a model, typed by a user, or constructed by another client. The server cannot assume they are correct merely because they passed through a host. It validates their shape, length, and allowed enum values.

The readiness score and recommendation are server-owned derived values. The view never recalculates them. That rule prevents two clients from reaching different conclusions and stops a DOM edit from turning a hold into a go decision.

The selected decision and note begin in the view because a person interacts with the form. They become authoritative only after the app-only tool accepts them. A local “success” state before a successful tool response is merely optimistic presentation.

The host owns the bridge and environment context. Theme, safe areas, and supported display modes are hints about presentation. They are not permission to perform a business mutation.

The model can help explain and route. It does not become an approval authority because it generated the initial tool arguments.

Write the allocation down:

Value	Origin	Authority	Used by
Release name	tool caller	validated server input	server and view
Check statuses	tool caller	validated server input	domain function
Score	domain function	server	text result and view
Recommendation	domain function	server	text result and view
Theme and safe area	host	host context	view only
Decision form	person in view	pending until server accepts	app-only tool
Decision receipt	server tool	server	view confirmation

This table is small, but it prevents a surprising amount of ambiguity.

Use a compact trust model

An MCP App touches several kinds of input. Treat each according to where it came from, not how plausible it looks.

Tool arguments are untrusted application input. Validate them on the server. A TypeScript type helps the author and editor; it does not inspect bytes at runtime. The schema supplied during registration is part of the protocol contract and should reject malformed values.

Tool results are trusted only to the degree that you trust your server implementation. The view should still parse defensively because versions drift and hosts can deliver errors or partial results. A missing array should produce an empty or error state, not a white panel.

Host context is presentation input. It may change while the view is mounted. Use it for spacing and colour preferences, never for authorisation.

User-entered notes are untrusted text. Render them as text, not HTML. If a future system stores them, validate length and apply the destination system's own authorisation and escaping rules.

URLs are data with executable consequences. This companion does not need external links in the view, which removes a whole class of allowlist and phishing concerns. If you add links later, validate the scheme and destination and use the host bridge rather than navigating the frame unpredictably.

Design for least capability

The best permission is the one the app does not request. Our view needs to receive tool input and results and call one named server tool. It does not need location, camera, clipboard, remote images, analytics, or direct database access.

That choice appears in several places:

- the resource CSP declares no remote connection or resource origins;
- the HTML bundle includes all JavaScript and CSS;
- the view contains no API key;
- the app-only tool has a narrow schema;
- the demonstration receipt contains synthetic identifiers only;
- logs describe lifecycle events without copying notes or check details.

Least capability is not just a security posture. It improves portability. Every external dependency is another host difference, CORS rule, privacy disclosure, and failure mode.

Separate reads from mutations

`review_release` is a read-like calculation. Given the same valid input and rule version, it returns the same result. It can be called by a model and safely repeated.

`record_release_decision` represents a mutation. In a real implementation it might write to a release service, create an approval record, or update a change-management ticket. It should not be exposed as another attractive model tool unless automated invocation is genuinely intended. The MCP Apps metadata lets us mark it for app visibility so the interactive flow can call it while reducing accidental model selection.

Visibility is not authorisation. A malicious or modified client may still attempt to invoke the tool. The server must authenticate the caller in a real deployment, check that the person may decide this release, validate the state transition, and use an idempotency key. Our local example cannot implement an organisation's identity policy, so it makes that limitation conspicuous.

Plan failure states before success

List failures alongside the happy path:

- The host has not connected to the app yet.
- The tool call is cancelled.
- The result contains an error.
- `structuredContent` is missing or from an incompatible version.
- The app-only call is denied by the host.
- The server rejects a decision.
- The view is torn down while a call is pending.
- A host does not support the preferred display mode.

Each failure should have a visible, honest outcome. “Connecting” is different from “No review supplied.” “Decision not recorded” is different from “Release blocked.” Preserve those distinctions so users know whether they should wait, correct input, or try another path.

For mutations, default to pessimistic confirmation. Disable the submit control while the request is pending, but do not claim success until a successful receipt arrives. If the call fails, keep the form data so the user can retry. Make the error specific enough to act on without exposing stack traces or secrets.

Keep the text path complete

The boundary exercise ends with the fallback. Write the text result format before styling the dashboard. A useful format for this project is:

```
Release readiness: Atlas 2.4.0
Recommendation: HOLD — 83/100
Blocking checks: Security review
Next action: Resolve the blocking checks (Security review) and rerun the review.
Rule version: 1.0.0
```

This tells a text-only user the subject, state, measurement, blocker, and next action. The dashboard can add scanability, status grouping, and a decision control, but the result is not trapped inside the iframe.

A small threat model

You do not need a ceremonial document. Record assets, boundaries, likely misuse, and controls.

Assets include the integrity of the readiness recommendation, the authority to record a decision, any future release metadata, and the user's trust in what the host displays.

Boundaries are the tool input, the server-to-host result, the host-to-view bridge, and the view-to-server tool call.

Likely misuse includes malformed arguments, a manipulated view trying to record a decision, result-version drift, unsafe rendering of text, and accidental expansion of the CSP when adding a dependency.

Controls include runtime schemas, a pure server calculation, app-only visibility plus real server authorisation in production, text rendering rather than HTML injection, capability checks, no external origins, and tests at each boundary.

The threat model also names what is outside the example: host authentication, organisation roles, persistence, audit retention, and deployment security. Naming an omission is more useful than implying that a tutorial solved it.

Boundary checkpoint

Before scaffolding, you should be able to say:

- The server owns the recommendation.

- The view owns temporary form state, not business truth.
- The host owns embedding and context.
- The model can propose calls but does not approve a release.
- Read and mutation tools have different exposure and validation needs.
- A text-only client gets a complete answer.
- The initial app needs no remote network origin and no secret.

With those constraints in place, implementation becomes a sequence of explicit contracts rather than a pile of components.

3 Scaffold a Reproducible TypeScript Project

A tutorial project should be boring to install. Reproducibility is part of the lesson: a reader must be able to distinguish an application mistake from a moving dependency or an undocumented build step.

The companion pins exact package versions in its lockfile and records the Node.js range it was tested with. That does not freeze the ecosystem forever. It creates a known baseline from which an upgrade can be deliberate.

The directory map

The project separates browser code, server code, and pure domain logic:

```
companion/
├── main.ts
├── server.ts
├── mcp-app.html
├── src/  contracts.ts, main.tsx, release.ts, schema.ts, styles.css
├── test/  contracts.test.ts, protocol.test.mjs, release.test.ts,
│        schema.test.ts
├── package.json
├── tsconfig.json, tsconfig.server.json, tsconfig.test.json
└── vite.config.ts
```

The server may import the domain module. The UI may import shared result types or a defensive parser. The domain module must not import React or the MCP server.

This makes the central calculation cheap to test and hard to accidentally couple to a transport.

Pin the runtime assumptions

Record three things before installing:

- the supported Node.js major range;
- the exact direct dependency versions;
- the command that produces the distribution artifact.

The companion uses the official MCP Apps SDK, the MCP TypeScript SDK, React, Zod for runtime schemas, Vite for the browser build, and Node's built-in test runner. Development packages such as type definitions and the single-file Vite plugin are not shipped as browser dependencies.

Why exact versions in a teaching project? A caret range can resolve to a newer minor release with a changed type or default between the author's run and the reader's first install. A lockfile reduces that ambiguity. When you intentionally upgrade, update the version log and rerun all three test layers.

Never interpret a lockfile as a security promise. Dependency advisories still matter, and production consumers need an update process. Reproducibility and currency are separate concerns.

Use two TypeScript configurations

The browser and server compile against different environments.

Browser code needs DOM types and uses bundler-style module resolution. It should not accidentally gain Node globals. Server code targets a supported Node.js runtime and may import `node:fs`, transports, and server packages.

A shared configuration can work, but it often hides boundary mistakes. The companion uses a browser-oriented tsconfig.json for type checking and a tsconfig.server.json for server-side output or declaration checking. Both enable strict mode. Unused variables and fall-through cases are errors, not hints.

The desired failure is early and clear. If UI code reaches for process.env directly, the browser configuration should object. If server code assumes a DOM type, the server build should object.

Bundle one HTML resource

An MCP App resource is easiest to distribute when its HTML, JavaScript, and CSS are self-contained. Vite builds the React entry, and a single-file plugin folds the emitted assets into the document.

The build config receives an explicit HTML input and writes to dist. Production builds disable inline source maps and minify the result. Development builds may keep inline maps for debugging. The server reads the resulting HTML file and returns it through registerAppResource.

This is not the only valid architecture. A larger app may justify declared remote assets. For this project, a single file proves an important invariant: the view can render without contacting an origin outside the MCP resource flow.

After the build, inspect the artifact. Search for http:// and https://. A licence comment or documentation string can contain a URL, so a match is not automatically a request. The goal is to confirm that there is no remote script, stylesheet, font, image, or fetch target.

Give every script one job

A useful script set is small:

```
{
  "scripts": {
    "build": "typecheck and build commands",
    "test": "compile and run node:test suites",
    "test:protocol": "build and run the in-memory protocol test",
    "verify": "run tests, build, and protocol verification",
    "start": "run the built stdio server"
  }
}
```

The companion spells out the real commands rather than leaving pseudo-code. npm run build must fail if either TypeScript side fails or if the UI bundle fails. npm test must be non-interactive and return a non-zero status for a failed assertion. npm start runs the built stdio process; it must keep protocol stdout clean rather than printing an HTTP endpoint.

Avoid a script named deploy in the tutorial unless the book teaches a specific reviewed deployment. Starting a local server and publishing an internet service are materially different actions.

Build before adding behaviour

Create the smallest page and server entry that compile. Then run:

```
npm ci
npm run verify
```

At the first scaffold stage, the test suite may contain one domain smoke test and one artifact assertion. That is enough to validate the pipeline.

Interpret failures in order:

1. Installation failures concern the runtime, registry, package metadata, or lockfile.

2. Type failures concern contracts and environment boundaries.
3. Unit failures concern behaviour.
4. Bundle failures concern browser entry points and asset handling.
5. Runtime connection failures concern transports and host integration.

Do not skip a type error and debug the iframe. Fix the earliest broken layer.

Keep configuration explainable

Copying a working quickstart is reasonable; keeping unexplained configuration forever is not. For every non-default option, know what invariant it supports.

For example:

- `strict` protects the contracts the view relies on.
- bundler module resolution matches Vite's browser transformation.
- Node module resolution matches server execution.
- a single-file plugin supports a self-contained resource.
- `emptyOutDir` behaviour is chosen so server and view artifacts do not erase each other.
- production source maps are off so internal source is not embedded accidentally.

If you cannot explain an option, look it up before changing it. Configuration cargo cults become production outages when an upgrade removes the accidental behaviour they depended on.

Add a version note

The companion README records:

- application version;
- MCP Apps SDK version;
- MCP TypeScript SDK version;
- React and build-tool versions;
- tested Node.js versions;
- the date of the last successful clean install, test, and build.

The manuscript source log records the official documentation revision inspected for this edition. This helps a reader decide whether a mismatch is likely to be conceptual or simply version drift.

Scaffold checkpoint

Do not proceed until these statements are true:

- Browser and server code are checked under appropriate TypeScript environments.
- A clean command creates one self-contained HTML file.
- Tests run without watch mode.
- The domain module has no protocol or UI dependency.
- The server can read the built resource from a deterministic path.
- Exact direct versions and a lockfile exist.
- Nothing has been deployed and no external service is required.

The project is now ready to implement the Tool plus UI Resource contract.