AI · CLAUDE TOOLS

Create an MCP App with json-render

Project setup, the catalog and registry, streaming AI-generated UI specs, and exposing the same components to AI agents over MCP — define components once, render anywhere.

Based on the LinkedIn Learning course "Create an MCP App with json-render" by instructor Eve Porcello, published 17 April 2026.

4 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: PROJECT SETUP & CONFIGURATION

Create an MCP App with json-render

Chapter 1 · Project Setup & Configuration

> 📺 Based on the LinkedIn Learning course **"Create an MCP App with json-render"** by instructor **Eve Porcello**, published **17 April 2026**. View the original course →

What Is JSON Render?

JSON Render is the core tool this course is built around — a pipeline that turns a plain-language prompt into a structured **JSON spec**, which is then rendered into real, live UI components. The idea splits neatly into two halves: an AI model decides *what* should appear on screen (expressed as JSON, not raw markup), and a separate rendering layer decides *how* that JSON actually becomes pixels. That separation is what makes the same spec usable in more than one place — a theme this course returns to directly in Chapter 4, once the same components get exposed to AI agents over MCP as well as to a browser.

The Project Stack

TOOL	ROLE IN THIS PROJECT
Next.js	The application framework — routing, API routes, and the React rendering layer
TypeScript	Typed JavaScript — catches spec/component mismatches at build time rather than runtime
Tailwind CSS	Utility-class styling for every rendered component
ESLint	Linting, keeping the generated and hand-written code consistent
Shadcn (Shad CN)	The actual component library the catalog draws from — real, pre-built UI primitives (cards, badges, stacks) rather than building each one from scratch
Vercel AI SDK	The library used to stream AI-generated output back to the client
Zod	Schema validation — used to make sure an AI-generated spec actually matches the shape the renderer expects before it's trusted

Key Packages

Alongside the standard Next.js scaffold, this project installs a small, purpose-built set of packages:

```
npm install @jsonrender/core @jsonrender/react-schema @jsonrender/shad-cn
npm install ai zod
```

- `@jsonrender/core` — the core spec-to-render engine
- `@jsonrender/react-schema` — the schema layer describing what a valid spec can contain
- `@jsonrender/shad-cn` — the Shadcn-specific bindings, connecting the schema's component types to real Shadcn components
- `ai` — the Vercel AI SDK itself, used for streaming model output
- `zod` — schema validation, used throughout to keep AI-generated data honest

Authentication: The Vercel AI Gateway

Rather than managing a separate API key for every individual model provider, this project authenticates through the **Vercel AI Gateway** — a single API key, stored in `.env.local`, that unlocks access to multiple different LLM models through one consistent interface.

```
# .env.local
AI_GATEWAY_API_KEY=your_key_here
```

WHY THIS MATTERS

A single gateway key means the model used later in Chapter 3's own API route (Claude Haiku 4.5) can be swapped for a different model without touching authentication code at all — the gateway, not the individual provider, is what the app actually talks to.

A STANDARD, WORTH-REPEATING CAUTION

`.env.local` should never be committed to version control — it's the file real API keys live in, and Next.js's own default `.gitignore` already excludes it for exactly this reason.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why JSON Render's approach — an AI model producing a JSON spec, with a separate renderer turning that spec into UI — is different from simply asking a model to generate raw HTML or JSX directly.

View solution

EXERCISE 2

Match each of the three `@jsonrender/*` packages (core, react-schema, shad-cn) to the specific real job it does in the pipeline, and explain why the project needs all three rather than just one combined package.

View solution

EXERCISE 3

A teammate suggests hardcoding the AI Gateway API key directly into `route.ts` instead of using `.env.local`, "to save a step." Explain, in your own words, why this is a bad idea.

View solution

CHAPTER 1 QUICK REFERENCE

- **JSON Render**: prompt → JSON spec → rendered UI, with generation and rendering deliberately kept separate
- Stack: Next.js, TypeScript, Tailwind, ESLint, Shadcn, Vercel AI SDK, Zod
- Core packages: `@jsonrender/core`, `@jsonrender/react-schema`, `@jsonrender/shad-cn`, plus `ai` and `zod`
- Auth via the **Vercel AI Gateway** — one API key in `.env.local`, multiple models

CHAPTER 2: BUILDING THE CATALOG & REGISTRY

Create an MCP App with json-render

Chapter 2 · Building the Catalog & Registry

> 📺 Based on the LinkedIn Learning course **"Create an MCP App with json-render"** by instructor **Eve Porcello**, published **17 April 2026**. View the original course →

Chapter 1 covered the stack and the packages. This chapter builds the first two real, concrete pieces the whole project depends on: the **catalog**, which says what components exist, and the **registry**, which says how each one actually renders.

The Catalog: What the AI Model Is Allowed to Use

`catalog/catalog.ts` defines the full, allowed set of components an AI-generated spec is permitted to reference — built directly from Shadcn's own component definitions. This is the real, concrete version of Chapter 1's own "constrained output" idea: the model isn't free to invent an arbitrary component name in its response, it can only choose from whatever this file actually lists.

```
export const catalog = {
  card: { /* Shadcn Card definition */ },
  badge: { /* Shadcn Badge definition */ },
  heading: { /* text-style definition */ },
  text: { /* text-style definition */ },
  stack: { /* layout definition */ },
};
```

> **WHY THIS IS ITS OWN FILE**
>
> Keeping the catalog separate from both the schema (Chapter 1) and the registry (below) means the "what's allowed" list can be reviewed, extended, or trimmed on its own — adding a new component to what the AI can reference is a one-file change, not a change scattered across the rendering logic too.

The Registry: Connecting Types to Real Implementations

`registry/registry.tsx` is where a component *type* named in the catalog (like `"card"` or `"badge"`) gets connected to the actual, real React component that renders it on screen.

```
"use client";

import { Card, Badge } from "@/components/ui";

export const registry = {
  card: Card,
  badge: Badge,
  // ...every catalog entry needs a real implementation here
};
```

THE REAL REQUIREMENT: "USE CLIENT"

`registry.tsx` needs the `"use client"` directive at the top of the file, making it a real client component in Next.js's App Router. Server Components (Next.js's own default) can't hold live references to interactive React components the way the registry needs to — the registry exists specifically to be handed real component references at render time, which is a client-side concern.

Catalog vs. Registry

	CATALOG	REGISTRY
Answers	What component types exist / are allowed?	How does each type actually render?
Consumed by	The AI model (indirectly, via the schema) and validation	The renderer, at the moment a spec is drawn to screen
File	`catalog/catalog.ts`	`registry/registry.tsx`, requires `"use client"`

Hands-On Exercises

EXERCISE 1

A new component, `"alert"`, is added to `catalog.ts` but the AI-generated spec that references it fails to render. Explain, in your own words, the most likely reason, based on this chapter's own catalog/registry distinction.

View solution

EXERCISE 2

Explain, in your own words, why `registry.tsx` needs the `"use client"` directive while `catalog.ts` does not.

View solution

EXERCISE 3

A teammate suggests merging the catalog and registry into a single file, "since they're always used together anyway." Explain, in your own words, what real flexibility this would give up.

View solution

CHAPTER 2 QUICK REFERENCE

- **Catalog** (`catalog/catalog.ts`) — the allowed set of components, built from Shadcn definitions
- **Registry** (`registry/registry.tsx`) — connects each catalog type to its real component implementation
- The registry requires `"use client"` — it holds live component references, a client-side concern in Next.js's App Router
- Every catalog entry needs a matching registry entry, or a spec referencing it can't actually render

CHAPTER 3: SPECS, THE API ROUTE & THE DASHBOARD BUILDER

Create an MCP App with json-render

Chapter 3 · Specs, the API Route & the Dashboard Builder

> 📺 Based on the LinkedIn Learning course **"Create an MCP App with json-render"** by instructor **Eve Porcello**, published **17 April 2026**. View the original course →

Chapter 2 built the catalog (what's allowed) and the registry (how each thing renders). This chapter connects them into a real, working feature — a spec, an API route that generates one from a prompt, and a component that streams the result to the screen.

The Dashboard Spec

`specs/dashboard.ts` is a real, concrete example of the JSON structure this whole pipeline produces — a plain object describing which components render and with what props, built entirely from Chapter 2's own catalog entries.

```
export const dashboardSpec = {
  type: "card",
  children: {
    type: "stack",
    children: [
      { type: "heading", props: { text: "Server Status" } },
      { type: "badge", props: { text: "Online", variant: "success" } },
      { type: "text", props: { text: "Last checked 2 minutes ago" } },
    ],
  },
};
```

The root is a `card`; its children are a `stack` layout, itself containing a `heading`, a `badge`, and a `text` element — every one of those type names has to exist in the catalog and have a real registry entry, or Chapter 2's own rendering chain breaks at that node.

The API Route: Generating a Spec From a Prompt

`app/api/generate/route.ts` is the real endpoint that takes a user's prompt and streams an AI-generated spec back to the UI, using `streamText` from the Vercel AI SDK with **Claude Haiku 4.5** as the model.

```
import { streamText } from "ai";

export async function POST(req: Request) {
  const { prompt } = await req.json();

  const result = await streamText({
    model: "claude-haiku-4.5",
    system: "Use 'card' as the root. Use 'badge' for status values.",
    prompt,
  });

  return result.toDataStreamResponse();
}
```

WHY CUSTOM RULES MATTER

The system rules passed alongside the prompt ("use card as root", "badge for status") aren't decoration — they're what steers the model toward specs that actually match the catalog's own real constraints and the app's own visual conventions, directly reducing how often a generated spec references something the registry can't render.

The Dashboard Builder Component

`components/generated/dashboard-builder.tsx` is the real, user-facing piece: a form collects the user's prompt, and the response streams back via a `useUIStream` hook.

1	2	3	4
Form	**useUIStream**	**Renderer**	**Fallback**
User types a prompt	Streams the API route's response	Draws the streamed spec, via Chapter 2's registry	Shows the static dashboardSpec while loading

WHY THE STATIC FALLBACK MATTERS

While the real, streamed spec is still arriving, the renderer displays the static `dashboardSpec` from earlier in this chapter instead of a blank screen — a real, deliberate UX choice: showing *something*

immediately, then replacing it once the genuinely AI-generated version has finished streaming in.

Hands-On Exercises

EXERCISE 1

The API route's system rules are removed entirely, leaving only the raw user prompt. Explain, in your own words, the most likely real consequence, connecting your answer back to Chapter 2's own catalog/registry material.

View solution

EXERCISE 2

Explain, in your own words, what a user would actually see, moment by moment, from the instant they submit a prompt in the Dashboard Builder to the instant the AI-generated spec finishes streaming in.

View solution

EXERCISE 3

A teammate suggests removing the static `dashboardSpec` fallback entirely, showing a blank screen while the real spec streams in, "to simplify the component." Explain, in your own words, the real UX tradeoff this would introduce.

View solution

CHAPTER 3 QUICK REFERENCE

- **Dashboard spec**: card root → stack layout → heading, badge, text — every type must exist in the catalog and registry
- **API route** (`app/api/generate/route.ts`): `streamText` + Claude Haiku 4.5, with real system rules steering the model toward the catalog's own constraints

- **Dashboard Builder**: form → `useUIStream` → renderer, with a static fallback shown while the real spec streams in

CHAPTER 4: MCP INTEGRATION FOR AI AGENTS

Create an MCP App with json-render

Chapter 4 · MCP Integration for AI Agents

> 📺 Based on the LinkedIn Learning course **"Create an MCP App with json-render"** by instructor **Eve Porcello**, published **17 April 2026**. View the original course →

Chapter 3 wired the catalog and registry up to a browser-facing UI. This closing chapter exposes the exact same catalog and components to **AI agents** instead, over MCP (Model Context Protocol) — no separate component system, no duplicated definitions.

The MCP Server

`mcp/server.ts` stands up an MCP server using `createMCPApp` from `@jsonrender/mcp`, communicating over **stdio transport** — the standard, simple way an MCP server talks to a connecting client over standard input/output rather than a network socket.

```
import { createMCPApp } from "@jsonrender/mcp";
import { catalog } from "../catalog/catalog";
import { dashboardSpec } from "../specs/dashboard";

const app = createMCPApp({
  catalog,
  specs: { dashboard: dashboardSpec },
  transport: "stdio",
});

app.start();
```

> **THE SAME CATALOG, REUSED**
>
> Notice the MCP server imports the exact same `catalog` object Chapter 2 built for the browser-facing pipeline — there's no separate, parallel definition of what components exist for the "AI agent" side of the app.

What Agents Can See

Once running, the MCP server exposes real, structured information an agent can query and act on:

- **The full component list** — every entry in the catalog, so an agent knows what it's allowed to reference
- **Initial state** — a starting point for whatever the agent is being asked to build or modify
- **Available specs** — existing, named specs (like `dashboard`) the agent can inspect or extend, rather than starting from nothing

Testing With the MCP Inspector

The real, standard way to test an MCP server directly — without needing a full agent client wired up yet — is the MCP Inspector:

```
npx @modelcontextprotocol/inspector
```

Running this connects to the local MCP server and lets you browse its exposed tools directly — the same real catalog, initial state, and specs an actual AI agent would see, inspectable by hand before ever connecting a real agent to it.

The Core Value: Define Once, Render Anywhere

> **TYING THE WHOLE COURSE TOGETHER**
>
> This is the real payoff Chapter 1 opened with: the same catalog, the same registry, the same specs serve two completely different consumers — a human, typing a prompt into Chapter 3's Dashboard Builder in a browser, and an AI agent, querying this chapter's MCP server directly. Neither consumer needed its own separate component system; both draw from the identical source of truth built in Chapter 2.

	BROWSER PATH (CH. 3)	AGENT PATH (CH. 4)
Entry point	`app/api/generate/route.ts`	`mcp/server.ts`

	BROWSER PATH (CH. 3)	AGENT PATH (CH. 4)
Consumer	A human, via the Dashboard Builder UI	An AI agent, over MCP
Shared source	The same catalog, registry, and specs	

Hands-On Exercises

EXERCISE 1

A new component is added to the catalog and given a real registry entry, but only the browser-facing dashboard picks it up — an AI agent connecting via MCP still doesn't see it. Explain, in your own words, the most likely cause, based on this chapter's own "same catalog" material.

View solution

EXERCISE 2

Explain, in your own words, why testing with the MCP Inspector before connecting a real AI agent is a genuinely useful step, rather than an unnecessary extra one.

View solution

EXERCISE 3

Summarize, in your own words, how each of this course's four chapters contributed to the final "define once, render anywhere" result — one sentence per chapter.

View solution

COURSE COMPLETE

- **mcp/server.ts**: `createMCPApp` from `@jsonrender/mcp`, stdio transport, reusing the exact same catalog and specs as the browser path
- Agents see the real component list, initial state, and available specs
- Test with `npx @modelcontextprotocol/inspector` before connecting a real agent

- The course's own core value: **define components once, render anywhere** — a human in a browser (Ch.3) and an AI agent over MCP (Ch.4) both draw from the identical catalog, registry, and specs built in Chapter 2
- **Create an MCP App with json-render is now complete — 4/4 chapters.**