@NGAF/RENDER · ENTERPRISE GUIDE

The Enterprise Guide to Generative UI in Angular

Agents that render UI — without coupling to your frontend

cacheplane.ai · 2026

Contents

CHAPTER 1

The Coupling Problem

The Coupling Problem

Every Angular developer who has integrated an LLM agent hits the same wall. The agent returns structured output—maybe a product recommendation, a data visualization, or a multi-step form. You need to render it. The obvious solution:

```
@Component({
  template: `
    @switch (output.type) {
      @case ('product') {  }
      @case ('chart') {  }
      @case ('form') {  }
    }
  `
})
export class AgentResponseComponent {
  @Input() output: AgentOutput;
}
```

This works. Ship it. Demo goes well.

Then the agent team adds a new capability. They want to render comparison tables. Your switch statement doesn't handle it. The agent emits valid output, the frontend shows nothing. Now you're in a deploy queue behind three other teams, waiting to add `@case ('table')`.

You've created a bidirectional dependency that will strangle your velocity.

The Dependency Graph You Didn't Want

The coupling runs both directions. The agent's output schema is constrained by what the frontend can render. The frontend's component library is constrained by what the agent might emit. Change either side and you break the contract.

This creates a coordination tax on every feature:

1. Agent team proposes new output type 2. Frontend team reviews, estimates component work 3. Component gets built, tested, merged 4. Agent team waits for frontend deploy 5.

Agent capability finally ships

In practice, steps 2-4 take weeks. The agent team starts working around limitations. They emit markdown instead of structured data because at least markdown renders. You lose type safety, styling control, and interaction capabilities.

Scale Multiplies the Problem

One agent, one frontend, one team—manageable. Now consider reality:

- Three agents sharing common UI patterns
- Two frontends (web app, internal dashboard)
- Separate teams for agent development and frontend platform

Your switch statements are now scattered across repositories. Each frontend has its own mapping logic. Agent teams don't know which frontends support which output types. Capability matrices live in Confluence pages that nobody updates.
When someone asks "can the agent render an approval workflow?"—the answer requires archaeology across four codebases.

Inverting the Dependency

The fix is architectural: agents emit UI specifications, not domain data. The frontend interprets specs through a registry, not a switch statement.

```
// Agent emits spec
{
  "component": "data-table",
  "props": { "columns": [...], "rows": [...] }
}
// Frontend interprets via registry
const registry = defineAngularRegistry({
  'data-table': DataTableComponent,
  'product-card': ProductCardComponent,
  // ...
});
```

New capability? Agent emits it. If the registry has a mapping, it renders. If not, fallback behavior. No frontend deploy required for agent changes. No agent prompt changes required for component refactors.

The registry becomes the contract. Version it. Document it. Let teams evolve independently.

The Standard Problem

A registry per frontend doesn't solve coordination—it moves it. You still need agreement on what `"component": "data-table"` means, what props it accepts, how deeply specs can nest.

Without a shared specification, you'll build three proprietary formats. Your agents will need frontend-specific prompt branches. Your component libraries will drift.

This is why the approach demands an open spec—a grammar for describing UI that agents can target and any frontend can interpret. Not a component library. Not a design system. A protocol.

The next chapter introduces that protocol.

CHAPTER 2

Declarative UI Specs & the json-render Standard

Declarative UI Specs & the json-render Standard

The Problem with Ad-Hoc UI Generation

When LLMs generate UI, the output format matters as much as the content. Without a formal specification, teams end up with brittle prompt engineering, framework-specific JSON schemas, and UI descriptions that break when models update or requirements change. The json-render specification solves this by defining a framework-agnostic standard for describing component trees as structured JSON.

Anatomy of a json-render Document

A json-render document describes a component tree using three primitives: component name, props, and children.

```
{
  "component": "card",
  "props": {
    "title": "Q3 Revenue",
    "variant": "elevated"
  },
  "children": [
    {
      "component": "metric",
      "props": {
        "value": 2847000,
        "format": "currency",
        "trend": "up"
      }
    },
    {
      "component": "sparkline",
      "props": {
        "data": [12, 19, 15, 25, 22, 30, 28]
      }
    }
```

```
  ]
}
```

The structure is intentionally minimal. Component names are strings—the rendering framework resolves them to actual implementations. Props are arbitrary JSON objects validated at render time. Children form a recursive tree of the same structure.

Why an Open Standard Matters

Framework Portability: A json-render document generated for Angular works identically in React, Vue, or any conforming renderer. Backend teams don't couple to frontend implementation choices. The same agent serves web, mobile, and embedded contexts.

LLM Prompt Stability: Models trained on consistent schemas produce more reliable output. When json-render becomes the canonical format across projects, prompt engineering compounds—patterns that work in one system transfer directly to others.

Community Tooling: Validators, visual editors, and testing utilities build against the spec rather than proprietary formats. Schema validation catches malformed output before it reaches the renderer.

Control Flow in the Spec

The specification handles dynamic rendering through reserved component types:

```
{
  "component": "$if",
  "props": { "condition": "{{ user.isAdmin }}" },
  "children": [{ "component": "admin-panel", "props": {} }]
}
```

Iteration uses `$for`:

```
{
  "component": "$for",
  "props": { "each": "{{ items }}", "as": "item" },
  "children": [
    {
      "component": "list-item",
      "props": { "label": "{{ item.name }}" }
    }
  ]
}
```

Computed properties use template expressions. The renderer evaluates these against a provided context object, keeping the spec declarative while supporting dynamic data binding.

Google's A2UI Extension

Google's Agent-to-UI (A2UI) specification extends json-render with agent-specific patterns. It adds constructs for streaming updates, tool call visualization, and interrupt handling—concepts that don't exist in static UI rendering but are fundamental to agent interactions.

A2UI defines how partial UI updates arrive during generation, how tool invocations surface to users, and how human-in-the-loop checkpoints integrate with component trees. The `@ngaf/render` package implements both specifications.

The @ngaf/render Implementation

The `` directive consumes json-render documents and instantiates Angular components:

```
import { Component, signal } from '@angular/core';
import { RenderSpecComponent, defineAngularRegistry } from '@ngaf/render';
import { CardComponent, MetricComponent, SparklineComponent } from
'./components';
const registry = defineAngularRegistry({
  card: CardComponent,
  metric: MetricComponent,
  sparkline: SparklineComponent
});


@Component({
  selector: 'app-dashboard',
  imports: [RenderSpecComponent],
  template: ``
})
export class DashboardComponent {
  registry = registry;
  uiSpec = signal(null);
}


The registry maps component names to Angular components. Unknown
components throw at render time—fail fast rather than silent
degradation.
```

Prompting for Valid Output

LLMs generate spec-compliant JSON when prompts include the schema and examples:

```
Generate a json-render document for a user profile card.
Schema: { component: string, props: object, children?:
array }
Available components: card, avatar, text, badge

Example output:
{"component": "card", "props": {"variant": "outlined"}, "children":
[...]}
```

Structured output modes (JSON mode, function calling) enforce syntactic validity. Schema validation catches semantic errors—referencing undefined components or invalid prop types.

The specification creates a stable contract. Agents emit it. Renderers consume it. Neither side knows how the other works.

CHAPTER 3

The Component Registry

The Component Registry

A render-spec document references components by string name. The registry resolves those names to Angular component classes at render time. Without this mapping layer, the open standard would be theoretical—the registry makes it executable.

Defining the Registry

`defineAngularRegistry()` accepts a record of component names to Angular component classes:

```
import { defineAngularRegistry } from
'@ngaf/render';
import { CardComponent } from
'./card.component';
import { ButtonComponent } from
'./button.component';
import { DataTableComponent } from './data-
table.component';
import { AlertComponent } from
'./alert.component';
export const uiRegistry =
defineAngularRegistry({
  'Card': CardComponent,
  'Button': ButtonComponent,
  'DataTable': DataTableComponent,
  'Alert': AlertComponent
});
```

The keys are the exact strings that appear in your render-spec `component` fields. The values

are the component classes themselves—not selectors, not factory functions. This directness means the registry is statically analyzable and tree-shakeable.

Providing the Registry

Two patterns, depending on your architecture.

Direct binding works for isolated cases where a single component owns the rendering context:

```
@Component({
  template: ``
})
export class PreviewComponent {
  registry = uiRegistry;
  spec = input.required();
}
```

Dependency injection suits applications where multiple components render specs against a shared registry:

```
// app.config.ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideRender({ registry: uiRegistry })
  ]
};
```

When both are present, the direct `[registry]` binding takes precedence. This lets you

override the global registry for specific rendering contexts—useful for sandboxed previews or A/B testing component implementations.

Resolution and Input Mapping

`` walks the spec tree, resolving each `component` string against the registry. For each node, it instantiates the corresponding Angular component and maps the spec's `props` object to `@Input()` bindings.

The mapping is direct: a prop named `title` binds to an input named `title`. No transformation, no case conversion. If your component expects `@Input() headerText` but the spec sends `header_text`, the binding fails silently—Angular's standard behavior for unknown inputs.

This is intentional. The registry defines *which* components exist; your component contracts define *what* they accept. Keep those contracts stable or version them explicitly.

Unknown Components

When a spec references a component name not present in the registry, `` renders nothing for that node by default. No error thrown, no console warning—just a gap in the output.

For development, enable strict mode through `provideRender({ strict: true })`. This throws

on unknown component names, surfacing mismatches immediately.

For production, consider a fallback component:

```
export const uiRegistry =
defineAngularRegistry({
  // ... your components
  '__fallback__': UnknownComponentPlaceholder
});
```

The `__fallback__` key is a convention, not a framework feature. Your error boundary strategy depends on your domain—some applications should fail visibly, others should degrade gracefully.

Registry Versioning

Specs persist. Registries evolve. The mismatch creates a versioning problem.

The cleanest solution: never remove component names from the registry. Deprecate by redirecting old names to new implementations:

```
export const uiRegistry =
defineAngularRegistry({
  'DataGrid': DataGridV2Component,      //
current
  'DataTable': DataGridV2Component,     //
legacy alias
});
```

For breaking changes in prop shape, version the component name itself (`CardV2`) or handle transformation inside the component. The registry stays stable; the component absorbs the complexity.

CHAPTER 4

Streaming JSON Patches

Streaming JSON Patches

Generative UI collapses the moment you wait for complete responses. A data table with 50 rows, each containing nested product details, might produce 15KB of JSON. Sending the full document on every update—when a single cell changes—creates unnecessary latency and forces the UI to block until the entire payload arrives. Users stare at spinners while the agent has already produced usable content.

@ngaf/render solves this with JSON Patch (RFC 6902), streaming incremental operations that

mutate the UI spec in place as the agent generates it.

The Problem with Full-Document Streaming

Consider an agent building a dashboard. The initial render spec might be 8KB. Adding a chart adds 2KB. Traditional approaches either:

1. Wait for the complete spec before rendering anything
2. Re-transmit the entire document on each change

Both approaches scale poorly. A spec that grows through 20 incremental updates would transmit 20 full copies—potentially hundreds of kilobytes for what amounts to a few patch operations.

JSON Patch (RFC 6902)

JSON Patch defines three core operations for mutating JSON documents:

- **add**: Insert a value at a path (`/dashboard/widgets/3`)
- **replace**: Swap a value at an existing path
- **remove**: Delete a value at a path

Each operation targets a specific JSON Pointer

location. The patch `[{"op": "add", "path": "/rows/-", "value": {"id": 42, "name": "Widget"}}]` appends a single row without touching the rest of the document.

Patch-Based Agent Output

Instead of emitting complete specs, the agent streams patch operations as it generates content:

```
{"op": "add", "path": "/rows/0", "value": {"id": 1, "status": "pending"}}
{"op": "add", "path": "/rows/1", "value": {"id": 2, "status": "active"}}
{"op": "replace", "path": "/rows/0/status", "value": "complete"}
```

Each line is independently parseable. The render layer applies patches immediately, updating only the affected portion of the component tree.

Partial JSON Parsing and Skeleton States

Real streams don't arrive in neat lines. TCP chunks split mid-token. @ngaf/render handles incomplete JSON by maintaining parse state across chunks, rendering valid portions while buffering incomplete fragments.

Skeleton states emerge naturally from this model. The agent can emit structural placeholders first—empty arrays, loading indicators—then fill them progressively:

```
  @Component({
    template: ``,
  })
  export class DashboardComponent {
    private store = signalStateStore({ rows: []
  });
    spec = this.store.state;
    registry = defineAngularRegistry({ DataTable,
  SkeletonRow });
  constructor() {
      this.streamPatches().subscribe(patch ⇒
  this.store.applyPatch(patch));
    }
  }
```

The `signalStateStore` from @ngaf/render manages immutable state updates. Each `applyPatch` call triggers fine-grained Angular signals, re-rendering only components bound to changed paths.

Performance Characteristics

Patch-based updates are O(change), not O(spec size). Appending one row to a 500-row table touches one array slot. The differ doesn't walk the entire spec; it applies the operation directly to the target path.

This matters at scale. A real-time monitoring dashboard receiving 10 updates per second would choke on full-document replacement. With patches, each update carries only the delta—typically under 200 bytes—and applies in microseconds.

The tradeoff is complexity at the agent layer. Your backend must track spec state and emit valid patches. But the rendering performance gains compound: faster time-to-first-paint, lower bandwidth, and UI that feels alive as the agent thinks.

CHAPTER 5

State Management & Computed Functions

State Management & Computed Functions

Static UI specs hit a wall fast. The moment you need a total that updates when line items change, or a button that disables based on form state, you're beyond what static JSON can express. Production generative UI requires computed properties and collection rendering—capabilities that @ngaf/render delivers through `signalStateStore()` and spec-level computed functions.

Agent-Managed State with signalStateStore()

`signalStateStore()` creates a reactive state container that both agents and components can manipulate. The agent initializes state through the spec, components update it via user interaction, and computed properties derive new values automatically.

```
import { signalStateStore } from '@ngaf/render';
const store = signalStateStore({
  items: [
    { name: 'Widget', price: 25, quantity:
2 },
    { name: 'Gadget', price: 40, quantity:
1 }
  ],
  taxRate: 0.08
});

const spec = {
  type: 'invoice',
  state: store,
  subtotal: { $compute: 'items.reduce((sum, i)
⇒ sum + i.price * i.quantity, 0)' },
  tax: { $compute: 'subtotal * taxRate' },
  total: { $compute: 'subtotal + tax' },
  lineItems: {
    $repeat: 'items',
    type: 'line-item',
    name: { $compute: '$item.name' },
    amount: { $compute: '$item.price *
$item.quantity' }
  }
};
```

The store exposes signals. When `items` changes—whether from agent streaming or user input—`subtotal`, `tax`, and `total` recompute. No imperative wiring required.

Computed Properties: Declarative Derived State

Computed properties use the `$compute` key to define expressions evaluated at render time. These expressions access the state store's current values and can reference other computed properties, enabling dependency chains.

The expression syntax is intentionally constrained JavaScript. It supports property access, arithmetic, array methods, and ternary operators—enough for UI logic, not enough to become a security liability. The renderer evaluates expressions in a sandboxed context with access only to the state store and iteration variables.

Repeat Loops for Collections

The `$repeat` directive iterates over arrays in state, rendering a component instance for each element. Within the repeated block, `$item` references the current element and `$index` provides the iteration index. This handles the common case of rendering lists, tables, and card grids without requiring the agent to enumerate every instance.

Drawing the Line: Spec Logic vs. Component Logic

Computed functions handle derived *data*. Components handle derived *behavior*. If you're calculating a display value—totals, formatted dates, conditional text—that belongs in the spec. If you're managing focus, coordinating animations, or handling complex validation workflows, that belongs in the component.

The heuristic: if an agent could reasonably want to change the logic, put it in the spec. If the logic is intrinsic to how the component works, keep it in the component.

Testing Computed Behavior

Test computed properties by manipulating the state store and asserting against the rendered output:

```
  it('should recompute total when items change',
  () => {
    const store = signalStateStore({ items: [{
  price: 10, quantity: 1 }], taxRate: 0.1 });
    const spec = { type: 'invoice', state: store,
  total: { $compute: 'items[0].price *
  items[0].quantity * (1 + taxRate)' } };

    const fixture =
  TestBed.createComponent(TestHost);
    fixture.componentInstance.spec = spec;
    fixture.detectChanges();

  expect(fixture.nativeElement.textContent).toContain(
  // 10 * 1.1
```

```
  store.update(s => ({ ...s, items: [{ price:
20, quantity: 2 }] }));
  fixture.detectChanges();


expect(fixture.nativeElement.textContent).toContain(
// 40 * 1.1
});
```

Drive the store, trigger change detection, assert the DOM. The reactive graph handles the rest.