User Blueprints, Blueprint Adapters, and a Global Inspector Field Registry (Conversation Notes v1)

Date: 2026-02-04
Scope: Design/architecture discussion (no code changes in this conversation)

This document captures a design discussion about making the app's template/blueprint system more flexible, specifically around:

- Repeatable components (monster icons, potential multi-row icons, stats blocks)
- Avoiding risky DB normalization/migrations in a browser-first app
- Introducing a "generic/composable" blueprint that can be augmented at runtime
- Persisting user-customised blueprint instances and promoting them into reusable user "templates"
- Replacing per-template inspector field mappings with a global Field Registry + global ordering

The overarching goal was to add meaningful flexibility while:

- staying **template-driven** (not a full canvas/layout editor),
- keeping rendering and export **deterministic**,
- avoiding "over-architecting" and risky client-side migrations,
- and keeping backward-compatibility options open.

Problem statement (as discussed)

The current hard-coded templates + blueprint-driven renderer cover most base-game HeroQuest cards, but users have shown interest in creating variants that don't fit neatly into the existing set of assumptions (single monster icon, fixed title placement, etc.).

The key product/engineering tension explored in this conversation was:

- Add flexibility for user-created card variants (repeatable elements, configurable

- Reworking the existing “drafts per template” workflow (cards don’t switch templates mid-edit).
- Solving stable identity for repeatables in a final form (explicitly noted as a future requirement/TODO).
- Exhaustively supporting every possible component type or per-template bespoke inspector ordering.

Glossary (as used in this discussion)

- **Template**: The product/editor definition of a card type (name/kind, what fields are editable, defaults, grouping in UI, i18n labels, etc.). Today this is keyed by `templateId` in `src/data/card-templates.ts`.
- **Blueprint**: The render plan for the SVG card output: ordered `layers` and optional `groups` (e.g. bottom stack) keyed by `templateId` in `src/data/blueprints.ts`, rendered by `src/components/BlueprintRenderer/index.tsx`.
- **Adapter / Proxy**: A runtime transformation step that takes a base blueprint and “augments” it (insert/move/remove/reorder elements, adjust bounds/origins) based on card properties and/or user configuration.
- **User blueprint**: A blueprint persisted to the user’s local DB/storage (not hard-coded), referenced by a card via `blueprintId`.
- **Custom template (user-facing)**: A saved, named, reusable user blueprint + metadata (name/kind). This is effectively “a user’s template”.

Non-goals (explicit or implied)

- No immediate code changes were requested in the conversation.
- No requirement to build a fully general “empty canvas” / drag-drop designer in this iteration.
- No requirement to support arbitrary/unbounded complexity (e.g. unlimited stats blocks) at the UI level; “repeatable in data” can still be constrained by business rules.
- No change to the current “drafts per template” workflow (template switching loads the last card/draft for that template; cards don’t change template mid-

Registries keyed by `templateId` (the "three maps" problem)

At a high level today there are multiple template-id keyed registries:

- `src/data/card-templates.ts`: template metadata and user-facing identity (name/kind, etc.)
- `src/data/blueprints.ts`: render layout (what appears where)
- `src/data/inspector-fields.ts`: per-template inspector field lists / ordering

The discussion aims to reduce duplication and make it possible for user-defined blueprints/templates to automatically provide a workable inspector and render plan.

Discussion timeline and conclusions

1) Template flexibility direction (constraint-driven composition vs full layout builder)

Initial framing:

- There are two product directions:
- Maintain curated, opinionated templates/blueprints that cover most use cases.
- Provide an "empty template" / layout composer where users add components and position them.
- A middle path was preferred:
- Templates become compositions of reusable components with constraints.
- A "blank/generic" template can exist as an advanced mode, but the core experience remains template-driven and predictable.

Key motivation:

- Add flexibility for real-world custom cards (e.g. multiple icons, different title positioning) without becoming a full design tool.

Compromise/workaround that avoids over-architecting:

- Prefer a constrained, template-driven approach that introduces a "generic/composable" option for advanced users, instead of turning the entire app into a full layout editor.

3) Migration/DB normalisation concerns (browser-first risks)

Concern raised:

- Fully normalising repeatables into relational tables increases:
- schema complexity
- import/export complexity
- risk of in-browser migrations failing or being interrupted (refresh mid-migration)

Preferred strategies:

- **Embedded arrays** (JSON shape) are the "sweet spot":
- avoids joins
- supports "unlimited" repeatables logically
- can be migrated safely with lazy/on-read conversion
- **Dual-read / lazy migration** pattern:
- If legacy single-id fields exist (e.g. `monsterIconId`), treat them as `monsterIcons[0]` when the array is missing.
- Optionally write only the new shape on save.
- Export/import accepts both shapes, with an explicit format version.

Outcome:

- Embedded arrays were accepted as the right approach (and consistent with an existing "double stats" feature already using embedded arrays).

---

4) Multiple stats blocks / "repeatables" in general (not fully committed, but planned for)

This topic came up as a parallel to monster icons:

- There is skepticism that users truly need multiple stats blocks, and physical card space will cap what's realistic.
- However, it was acknowledged that if the system supports repeatables properly, business rules (caps) can live above the data model.

Compromise suggested:

- Use repeatable data structures (arrays) where appropriate, but keep UI

5) Title flexibility (none/top/bottom + ribbon/none)

Observed competitor behavior:

- Title position: `none | top | bottom`
- Title style: `none | ribbon` (and possibly “plain text” variants)

Current implementation reality:

- Title is a top-level blueprint layer today (not inside the bottom stack group).
- Some templates (hero/monster) use a bottom stack group for icon/stats/text.

Key issue identified:

- “Bottom title” is mostly a layout collision problem:
- It can overlay artwork/stack unless the layout reserves space or the title participates in stacking.

Rationale:

- This is a classic example where “just add a property” can force a lot of bespoke per-template math unless the blueprint system has an explicit composition model.

Three conceptual approaches were discussed:

- **Slot-based**: blueprint defines a top slot and bottom slot; card chooses which slot is active.
- **Group-based**: title becomes a group child so it participates in stacking and pushes other content.
- **Blueprint variants**: separate blueprints for each title configuration.

The discussion leaned toward:

- Introducing a **generic/composable blueprint** designed to support title movement via an adapter:
- some templates/blueprints won’t have a title at all and shouldn’t be forced into the new model
- the generic blueprint is explicitly “composed on the fly”

---

7) Templates vs blueprints (and what each is responsible for)

Clarification reached:

- Templates are not *only* used to choose a blueprint; they also define editor/product metadata.
- Blueprints define rendering layout and binding to data keys.
- Today both are keyed by `templateId` and therefore "template selection implies blueprint selection".

Additional evolution proposed:

- Over time, some of what's currently in template metadata may become "user-defined" via saving user blueprints/templates:
- Users can save a blueprint as a reusable "template" with a name and kind.
- Built-in templates/blueprints remain hard-coded and i18n-capable.

However:

- It was argued (and accepted) that user-facing identity (name/kind) should remain "template-like metadata", not embedded into the rendering blueprint itself, even if the code currently blurs these concepts.

---

8) Global Field Registry (replace per-template inspector field lists)

Key idea:

- Replace `src/data/inspector-fields.ts` per-template mappings with:
- a **Global Field Registry** (field key → UI semantics)
- a **Global Order** (single ordered list of keys/sections)
- blueprint-driven inclusion (if blueprint binds a key, show that editor)

Blueprint introspection:

- Determine "used keys" by scanning blueprint binds (`titleKey`, `textKey`, `imageKey`, `iconKey`) across layers and groups.

Important nuance:

blueprint record saved in the DB).
- Card changes to layout are saved by updating the referenced blueprint record.

Promotion to reusable “custom template”:

- A user can “Save blueprint as template” (name + kind), enabling use across cards.
- When a blueprint is referenced by multiple cards, it becomes **locked**:
- editing should prompt to fork (save as new blueprint) rather than mutating the shared blueprint
- If a blueprint is referenced by exactly one card, it remains mutable for a smooth UX.

Future optional nuance mentioned:

- A “fluid/shared” blueprint mode could allow safe edits that affect all cards (e.g. reordering without removing components), but this was explicitly deferred.

---

10) Dedupe / “this blueprint already exists”

Concern:

- Users may end up creating many blueprint-per-card instances; duplicates may exist.

Options discussed:

- Avoid surfacing “memory management” to users as a primary concern.
- Internally compute a blueprint hash (canonical JSON) to:
- silently dedupe identical blueprints, or
- show a soft prompt only when it’s relevant (e.g. during “Save as template”: “this matches existing template X; reuse?”)

Outcome:

- Not required for v1, but storing a hash is a low-risk foundation.

This was explicitly called out as a reminder to capture in requirements when documenting the work.

Consolidated “Agreed Model” (as of this conversation)

Primary vs secondary fields

- **Primary bound fields** indicate presence:
- Examples: `title`, `description`, `imageAssetId`, `monsterIconRows`, stats blocks
- **Secondary option fields** modify behavior when the primary exists/ is used:
- Examples: `titleStyle`, `titlePosition`, `monsterIconLayout` (per row)
- Inspector and renderer visibility require **both**:
- the card data supports/contains it, and
- the blueprint binds/uses it.

Repeatable monster icons

- Canonical model is “rows”:
- `monsterIconRows: Array<{ icons: IconRef[]; layout: Layout }>`
- Renderer binds rows by occurrence index of `MonsterIconRow` components in the blueprint.
- Inspector uses one editor that displays all rows.

User blueprint persistence

- Cards: `templateId XOR blueprintId`
- Blueprints:
- mutable while referenced by one card
- locked when shared; edits fork into a new blueprint

Adapter

- A deterministic, pure (non-mutating) transformation from base blueprint + config/card data → resolved blueprint.
- Resolved blueprint retains the same persistent blueprint ID.

Open questions / “Decide later” list

1. **Stable IDs for rows/icons**: when/if to add persisted IDs to repeatable items.
2. **Title bottom behavior**: overlay vs consume space (group-based stacking vs layer slot) for the generic blueprint.
3. **Blueprint dedupe policy**: silent internal dedupe vs “reuse existing” prompts (likely only in save-as-template flow).
4. **Export/import**: exact file format changes to include user blueprints + metadata, and backward-compat reading.
5. **Capability inference vs explicit meta**:
6. Many things can be inferred from blueprint binds.
7. Some constraints (max rows/icons, optional features) might need explicit blueprint/template metadata if they become template-specific.

Suggested next documentation step (when ready)

Create a requirements/spec doc for:

- “User Blueprints + Global Inspector Registry”
- “Monster Icon Rows (repeatable component)”
- “Generic/Composable Blueprint + Adapter”

Include explicitly:

- The `templateId XOR blueprintId` rule
- The adapter’s deterministic/pure requirement and what it is allowed to transform
- The primary/secondary field inclusion rule
- Missing asset placeholder behavior
- A note/TODO about stable IDs for repeatables