ChatGPT

Standard UMG Citadel Repository Scaffold

The **NeoUMG Citadel** repository is structured for clarity and easy expansion. It separates core components into logical directories and provides templates and documentation for building **standard (non-Web3) UMG** applications. This scaffold is designed to be zip-ready and **Bolt-friendly**, allowing one-click integration into the Bolt builder environment. Below is the proposed structure and contents:

Core Directory Structure

- `/data/` – Contains all data files, block templates, presets, and project plans. This is where the building blocks of UMG apps live.
- `/public/` – Static public assets (e.g. icons, images) meant to be served with the app. It includes an `icons/` subfolder for any SVG or image assets.
- `/src/` – Source code utilities or scripts for the Citadel system. For example, a utility for merging blocks (merge engine) resides here.
- `/docs/` – Markdown documentation for developers and operators. It explains the schema, color system, engine logic, import/export formats, and the Citadel flow.

Each directory is organized to keep the repository clean and intuitive. For instance, all JSON templates are under `/data`, all documentation under `/docs`, etc., making it straightforward to locate or add new components.

Organizing Block Templates by Category

All JSON **block templates** are stored under `/data` in subfolders by category. This categorization mirrors the major domains of UMG blocks (as identified in the UMG plans) – for example:

- `/data/templates/business/` – Business plan builder blocks (e.g. strategy, finance, HR, legal) [1]
- `/data/templates/webapp/` – Website development blocks (e.g. frontend UI components, backend logic, auth) [1]
- `/data/templates/chatbot/` – Chatbot logic blocks (e.g. prompt flows, personality modules, tool integrations) [1]

Additional categories can be added as needed (for instance, if there were other domains like `education`, `game`, etc., those would get their own subfolders under `/data/templates/`). Each template file is a `.json` following the unified UMG Block Template schema (see **UMG_BLOCK_TEMPLATE.md** for the full reference). All template files use placeholder values where appropriate (like `<BLOCK NAME>`, `<CATEGORY>` placeholders) to make them easy to customize for new apps.

Unified Block Template Format: Every block template JSON shares a common structure with standard fields. For example, a typical block template includes keys such as `block_id`, `label`, `category`, `description`, and others for metadata and behavior [2]. A simplified schema (filled with placeholders) looks like:

```
{
  "block_id": "auto_generate_or_use_prefix",
  "label": "<BLOCK NAME>",
  "category": "<CATEGORY>/<SUBCATEGORY>",
  "description": "A short summary of what this block does.",
  "editable_fields": { ... },
  "molt_type": "Primary | Instruction | Philosophy | Subject",
  "tags": ["example", "<CATEGORY>"],
  "canto_overlay": { ... },
  "ledger": { ... },
  "trigger": { ... },
  "export_config": { ... }
}
```

All block templates adhere to this format for consistency [2]. The `molt_type` field classifies the block's purpose (Primary, Instruction, Philosophy, or Subject) [3], and the templates are filled with dummy or placeholder content that a developer can replace with real values when creating a new block. By organizing templates by domain and using a uniform schema, operators can quickly find relevant blocks and ensure they all conform to UMG standards.

Sleeve Presets (`/data/sleeve_presets/`)

The Citadel introduces the concept of **sleeves** – higher-level presets or configurations that group stacks of blocks into a cohesive context. To accommodate this, we create a new folder for sleeve definitions. All sleeve preset JSON files (which define pre-configured block stacks or styles) reside in `/data/sleeve_presets/`.

For example, the file `bolt_color_sleeve_stack.json` (previously in the root or elsewhere) will be moved into `/data/sleeve_presets/bolt_color_sleeve_stack.json`. This file likely defines a preset "sleeve" – possibly a styled stack of blocks with a particular color scheme or layout. By placing it in `sleeve_presets`, we keep such config files separate from the generic block templates.

Organizing sleeve presets separately ensures that any pre-built configurations (like color themes or common stack arrangements) are easy to find and manage. If additional sleeve presets are created (for different UI themes or starter stacks), they would also live in this folder.

Comprehensive Documentation (`/docs`)

A set of Markdown documents under the `/docs` directory will provide clear guidance on using and extending the UMG Citadel system. These docs are crucial for clarity and serve as a reference for developers or operators interacting with the repository:

- `UMG_BLOCK_TEMPLATE.md` – **UMG Block Template Schema Reference.** This document describes the schema for block JSON templates in detail. It will list each field (such as `block_id`, `label`, `category`, `editable_fields`, `molt_type`, etc.) and explain its purpose. For example, it will note that every block has a human-readable `label`, a `category` for logical grouping, and a

`molt_type` that indicates the block's role [3]. It should also explain how placeholder values in templates should be replaced when creating actual blocks. Essentially, this is the blueprint for what a block JSON contains and how to fill it out properly.

- `COLOR_SYSTEM.md` – **Color Coding System.** Documents the standardized color scheme used in Citadel for different block types or purposes. For instance, primary blocks might be indicated with a **blue** color, subject-matter blocks with **green**, etc., to provide visual cues in the UI. This doc will outline each color and its assignment. (E.g., *Primary = Blue, Subject = Green, Instruction = Orange, Philosophy = Purple* as a potential scheme.) The goal is to maintain a consistent look and feel across the app by using these colors for block highlights, sleeve banners, or tags. By defining it in one place, all developers know which colors to use for new block types to keep the system uniform.

- `MERGE_ENGINE.md` – **Merge Logic & Engine.** Describes the merging logic used by Citadel's **merge engine**. The merge engine (implemented in a stub JavaScript file in `/src/utils/mergeEngine.js`) handles how blocks or their outputs are merged or combined when assembling the final application. This could involve merging JSON outputs, overlaying configurations, or snapping UI components together. The doc will cover the fundamental logic (for example, how blocks designated as mergeable can combine their content or how the engine decides to integrate one block's output with another). This corresponds to the system's block "merge" capability – as mentioned in the UMG plans, blocks can be snapped together and some are **mergeable** as groups [4]. The documentation ensures that anyone looking to modify or utilize the merge engine knows the intended behavior and placeholders (with notes that the actual code is stubbed out as a starting point).

- `VAULT_IMPORTS.md` – **Vault Plan Import Format.** Specifies the `.vault.json` format used for importing/exporting entire app plans (the "vault" files). A `.vault.json` is essentially a blueprint of an application composed of multiple blocks and possibly multiple sleeves/stacks. This doc will define how a vault file is structured: for example, it may include metadata like app name and version, and lists of sleeves or stacks with references to block IDs or categories. It will explain each part of the JSON (e.g., a section listing which block templates to instantiate for a given app, how they are grouped into stacks, and any connections or order between them). By providing this spec, developers can hand-edit or programmatically generate `.vault.json` files and be confident they will be understood by the Citadel system. (Since this is a standard non-Web3 environment, the vault spec focuses on off-chain use – essentially a JSON configuration of the app structure – as opposed to any on-chain references.)

- `CITADEL_FLOW.md` – **Citadel Flow: Sleeve > Stack > Block.** An overview document that explains the hierarchical relationship between **sleeves**, **stacks**, and **blocks** in the UMG Citadel architecture. It will likely clarify that:

  - A **Block** is the smallest unit of functionality or content (represented by a JSON template as above).
  - A **Stack** is an ordered collection of blocks assembled to work together (for example, a sequence of UI components or logic steps that form a feature). Blocks in a stack might execute or render in sequence or as a group.
  - A **Sleeve** is a higher-level container or context that can hold one or more stacks (think of a sleeve as a full module or a page that encompasses various block stacks). It might also carry a theme or state that applies to its stacks. Sleeves help organize the app into sections or phases.

This doc will walk through the flow of how an app is constructed: e.g., *"Inside a Citadel, sleeves are the top-level sections of an app. Each sleeve contains stacks of blocks. Blocks are snapped into stacks, and stacks fit within a sleeve to form a complete feature or page. When the app runs (or is built), blocks execute within their stack order, stacks ensure logical grouping, and sleeves manage the overall context or styling."* It provides a conceptual map so that anyone reading can visualize how a **block** comes to life as part of a **stack**, and how stacks live within a **sleeve**. This high-level flow is crucial for understanding how to design new templates or assemble an application using the repository.

All documentation is written in clear Markdown, with diagrams or examples as needed, to make adoption easy. The docs ensure that even newcomers can understand the UMG Citadel system structure, the role of each file, and how to extend it. Each doc focuses on one aspect (schema, color, merge logic, etc.) to keep information organized.

Repository README

A top-level **README.md** is included at the root of the repository to introduce and guide users. The README will cover:

- **Purpose of the Repository:** Explaining that this repo is the **Citadel Standard** for Universal Modular Generation (UMG) apps. It outlines how the structure provides a modular library of blocks and templates to rapidly build web applications (non-Web3 focus). Essentially, it's a base kit for creating or exporting UMG-based projects.

- **How to Use (Operator/Bolt Workflow):** Instructions for an *Operator* or the *Bolt* system on using this repository. For example, it will describe how an operator can import a `.zip` of this repo into the Bolt UI. Since the structure is Bolt-friendly, the README will note that Bolt can detect the `/data` templates and `/docs` to assist the user. It might say: *"To start a new app, open this repository (or import the zip) in the Bolt builder. Use the example.vault.json in* `/data/project_templates` *as a starting plan or create a new .vault plan. Bolt will load the block templates from* `/data/templates/*` *and apply any sleeve presets from* `/data/sleeve_presets/`*."* It will also detail how Bolt or an operator can **export** an app: presumably by selecting blocks and then generating output code (the structure is already set up to export to Replit or similar [5]). Essentially, the README acts as a quick-start guide for integrating with Bolt: from import, through editing, to export/deployment.

- **Where to Find Things:** A breakdown of the repository contents so users know where to add or edit components. E.g.:

- *"To add a new block template, put its JSON file in the appropriate folder under* `/data/templates/` *(or create a new category folder if needed)."*
- *"To modify or read about the block schema, see* `/docs/UMG_BLOCK_TEMPLATE.md`*."*
- *"Sleeve presets can be found in* `/data/sleeve_presets/` *– you can add new preset JSON files there."*
- *"Utility scripts (like the merge engine) are in* `/src/utils/`*."*
- *"Static assets for your web app (icons, images) go into* `/public/` *(you can add subfolders as necessary, e.g., for icons or images)."*

This way, a developer knows exactly where to place new files or find existing ones, maintaining the organized approach.

- **Contribution and Expansion:** Since the repo is meant for expansion, the README can also encourage best practices for adding new templates or docs. It can mention that the structure is akin to a template itself which can grow with more blocks or categories as UMG evolves, and that using GitHub for version control ensures that changes to templates or docs are tracked.

Finally, the README will highlight that this repository is optimized for quick setup: by following the README, an operator can get a UMG app environment up and running in Bolt with minimal hassle (truly *"one-click integration"* in ideal cases).

Optional Placeholder Files and Extras

To make the scaffold immediately useful and demonstrate how each part works, a few placeholder files are included:

- `/src/utils/mergeEngine.js` – a stub JavaScript file for the merge engine. This file contains a placeholder function or class (with documentation in comments) that represents how block merging might be handled. For now, it can be a no-op or a simple example (e.g., a function that takes two JSON blocks and merges their editable fields, or combines their outputs). Including this file indicates where and how the actual merge logic would be implemented. It ties in with **MERGE_ENGINE.md**, which explains the intended logic. Developers can flesh this out later; having it there shows the complete picture of the system.

- `/data/project_templates/example.vault.json` – a stub plan file for demonstration. This JSON might outline a very simple app (perhaps a combination of a few blocks from each category) to illustrate the format. For example, it could contain a JSON structure like:

```
{
  "project_name": "Example App",
  "sleeves": [
    {
      "name": "Main Module",
      "stacks": [
        { "name": "Intro Flow", "blocks": ["block_welcome.json",
"block_login.json"] },
        { "name": "Feature X", "blocks": ["block_featureX_step1.json",
"block_featureX_step2.json"] }
      ]
    }
  ]
}
```

(The actual schema will be defined in VAULT_IMPORTS.md.) This file serves as a template that users can copy and modify to create their own `.vault.json` for a new project. It also allows testing the

import process in Bolt or other tools – the example can be imported to generate a sample app structure.

- `/public/icons/` – an icons directory, possibly containing one sample SVG icon (e.g., `logo.svg` or a generic icon). This is mostly to illustrate the presence of static assets. It can be left mostly empty (with just a placeholder file or README) to show where to put public assets. If an example icon is included, it provides a reference for how an icon file is placed and can be referenced by blocks or exported site templates.

These placeholders are optional but recommended. They ensure that the repository is not empty in critical places, and they exemplify how to extend the system. For instance, `example.vault.json` ensures the `project_templates` folder is not empty and gives users a starting point to follow for their own plans. Likewise, `mergeEngine.js` stub and an icon file show the expected file types and naming conventions.

By including these, we aim for a **"batteries included"** approach: the structure is ready to use out-of-the-box, and everything is in place for a user to start experimenting or building with minimal setup.

Conclusion

The finalized scaffold provides a clean, well-documented foundation for the UMG Citadel system. It emphasizes **simplicity** (clear separation of concerns in directories), **documentation clarity** (extensive docs in `/docs` for every key aspect), and **integration readiness** (works with Bolt's import/export with one click, and is structured for GitHub version control). An operator or developer can download this repository as a ZIP, open it in the Bolt builder, and immediately have access to a library of block templates, preset configurations, and guidance on how to assemble and deploy an app.

This organization not only helps in the current state (standard non-Web3 apps) but is also designed to expand. New block categories, additional templates, or future Web3 enhancements can be added without disrupting the existing structure. All in all, the repository is **production-ready** to serve as the Citadel for UMG – a stronghold of modular blocks and knowledge, primed for building the next app with modular efficiency and ease.

---

[1] [4] [5] Umg App Plans.docx
file://file-3iHZ4jEp5NjJ7d2fkus5J4

[2] [3] Umg Block Metadata Policy.docx
file://file-32WqpT8NhTyHpBydDLTVf4