Token Orchestration

Intro & Terms

The current DTCG specification (as of Mar 2024) is primarily concerned without outlining the token types (8) and composite token types (9). To avoid using the term "tokens" too much, I'll refer to these together as "**data structures**." But an open question is how the DTCG spec allows composition and implementation of those data structures. Many design systems have the concept of "modes" (e.g. light mode vs dark mode) and "themes" (e.g. MUI Themes or Tailwind Themes), and tools have varying concepts, such as Figma's collections and modes. This "higher layer" of composing data structures is what I'll refer to in this document as "**orchestration**" as it's a question of not only how these data structures are *organized* but also how they are *utilized* in a working design system.

Other than the terms "data structures" and "orchestration," there are no other repeating terms that have the same definitions. I'll be referring to "modes," "themes," "collections," and more only to borrow terms defined by specific tools and implementations. They have no shared definitions, and will conflict with one another.

This document is a guide to different notable terms and patterns that exist in token orchestration.

Approach 1: Linear

The **linear** approach to token orchestration treats modes as the children of tokens, often appearing as optional metadata. This results in every token having a "default" value, and only some tokens extending that with mode-specific values.

Note that the order inversion—tokens being children of modes—isn't really a consideration. It would make scanning tokens far harder which are currently foundational to everything.

Hierarchy

- **Token Manifest**
    - **Group**
        - **Token**
            - **Mode** (optional)

Examples

- Supernova (uses the term “theme” which operates like a mode)
- Specify

Approach 2: Matrix

The **matrix** approach to token orchestration puts tokens and modes on the same level, and examines all the combinations of each. For every mode, a token must have a value, and vice-versa. It’s often represented as a spreadsheet (matrix) for viewing and editing.

And though tokens can exist in hierarchical groups, there exists another dimension called **collections** that serve as a boundary for modes. In other words, modes don’t reach across the entire token manifest; they only exist within an individual collection. Collections do not share tokens (or token groups).

Hierarchy

- **Token Manifest**
    - **Collection**
        - **Group**
            - **Token**
        - **Mode**

Examples

- Figma

Approach 3: Resolver

The **resolver** approach is less of a hierarchy and more of a proposal of putting tokens through ordered stages to a final state. This changes the current JSON format into more of a Domain-specific Language (DSL) with a runtime (almost like a bundler). The token values can’t be ascertained without a “build” process (in a sense).

Hierarchy

Token Manifest → **Tokens** → **Modifiers** → **Sets** → (result)

Examples

- Tokens Studio

Considering all approaches

Evaluating the approaches isn't straightforward. Here are some key points to evaluate them on:

Pros/cons

	Pro	Con
Linear	DRY	Hard/impossible to validate
	Fallbacks built-in to the design	Hard to determine modes + mode values
	Works with current spec	
Matrix	Built-in validation	Unnecessary repetition
	Modes have a better hierarchy	No concept of "default" values
	Frontloading permutations can result in faster static analysis, and faster tooling	Changing one mode could rearrange the entire token manifest
Resolver	DRY	Token values aren't statically-determinable without a "build" step or "runtime"
		Essentially creates a DSL
		Possibly wouldn't work for many setups (if the "stages" are fixed)

Permutations

@gossi called out in a GitHub comment the problem of *permutations*. The linear approach takes the DRY stance of modes either having an "override" value; otherwise, use the default. The matrix approach enforces all permutations are frontloaded, which can result in undue repetition. While from a systems point of view, the end result is the same—all permutations of tokens & modes have to be evaluated at some point. But whether or not that is frontloaded in the tokens manifest, or just handled automatically by the tooling, is the question.

Static Analysis vs Runtime

Implementing any runtime component to the schema (including operations) would change the current nature of the DTCG to move from the realm of static analysis where it is currently (akin to JSONSchema or OpenAPI), to effectively becoming a Domain-specific Language (DSL) (or, at least, a language server). This includes @SorsOps's resolver proposal, and even @universse's applying themes proposal. It's worth deciding on whether **static analysis** is a core principle of the DTCG spec as it is with other specs (like JSONSchema)

Layering + Scoping

Another problem is "mode scoping," an issue that's been talked about frequently in the DTCG GitHub and elsewhere, but something no proposal addresses. Many design systems implement some variation of base → semantic → component token layers (example). But because there are no restrictions over whether modes cross these "layers" or not, it can lead to confusion and churn.

As a practical example, pretend you have a "base" grayscale from `100` – `1300` (values don't matter), and you decide your `semantic.text-color` token is `gray.700`. This ends up being "light mode," and when you add "dark mode," you use `gray.600` to get roughly the same contrast. Shortly on, you realize that your dark mode `gray.600` needs to be adjusted to improve contrast—you need a color that doesn't exist in gray `100` – `1300` at all. So the question becomes: does `semantic.text-color` end up being `gray.700` for light and dark modes, and instead `gray.700` gets light and dark modes? Or does that go against the idea of "stateless" core tokens, so we create a separate scale of `gray-light.XXX` and `gray-dark.XXX`, so light and dark modes can exist on the semantic layer)? The former answer is more "clever" but blurs the line between the core and semantic layers as the core takes on more "awareness." The latter answer preserves the layer boundaries better, but introduces the idea that there are "rules" over when core tokens can/can't be used in certain contexts (arguably also blurring the lines, but in a different manner). Either answer will "fork" the design system in 1 of 2 ways that is difficult/impossible to reverse down the line.

A hand-wavy answer to this is "it's up to every team to decide." But when we consider every design system implementing modes faces this same confusion, and it stems directly from how teams define modes/theming (i.e. the purpose of this document), this may not be a difference of opinion so much as it a symptom that modes/theming are ill-defined in any token system (and could probably be remedied with a more opinionated solution).

Appendix

Relevant Community Discussions

- Native modes and theming (#210)
- Token operations (#224)

Theming

Definitions

- Tailwind: Themes = color modes
- Material: Schemes = color modes (but multiple modes *may* exist on-page in some conditions)
- GitHub Primer: Themes = color modes
- IBM Carbon: Themes = color modes

Scope

- Tailwind: color, typography
- Material: color
- IBM Carbon: color
- GitHub Primer: color