ChatGPT

Tagaddod Design System: Structure and Publishing Guide

Design System Structure and Multi-Brand Support

Monorepo Architecture: The Tagaddod Design System is organized as a monorepo (using Turborepo) with separate packages for tokens, themes, components, and documentation [1] [2]. Key packages include:

- `@tagaddod/tokens` – Design tokens (colors, spacing, etc.) following W3C design token standards [3] [2].
- `@tagaddod/themes` – Brand theme configurations (for Tagaddod vs. GreenPan) [2].
- `@tagaddod/react` – The React component library built with TypeScript and React 19 [4] [2].
- `@tagaddod/storybook` – The Storybook-based development and documentation environment [5] [2].

This structure enables **multi-brand theming** and **internationalization (LTR/RTL)** out of the box [6]. The design system supports both Tagaddod and GreenPan brand styles and Arabic (RTL) in addition to English (LTR) without changing component code [7].

Design Tokens and Themes: All design values (colors, fonts, spacing, etc.) are defined as JSON tokens in the `@tagaddod/tokens` package. The token source is organized into **core primitives**, **semantic aliases**, and brand- or locale-specific overrides [8] [9]. For example, core tokens define base colors and sizes, semantic tokens map those to intended uses (e.g. `color.text.primary` referencing a gray value), and brand override files adjust specific values for GreenPan or Tagaddod as needed [10] [11]. There are also locale overrides (e.g. if needed for RTL-specific spacing or font adjustments) under a `locales` directory [12].

The tokens are built using *Style Dictionary* to generate multiple formats: **CSS custom properties**, **SCSS variables**, and **JavaScript constants** [13] [14]. During the build (`yarn build` in `packages/tokens`), Style Dictionary reads all token JSON files, resolves references (e.g. `{color.green.500}`), and outputs compiled token files [15]. The output includes a master CSS file of custom properties and specialized files per brand and locale. For example, after build the `dist` folder contains: a global `tokens.css` with all `--t-*` custom properties, and subfolders for each brand (`tagaddod/vars.css` and `greenpan/vars.css`), further divided by locale (e.g. `tagaddod/en/vars.css`, `tagaddod/ar/vars.css`) [16] [17]. Each brand's CSS variables override or extend the base tokens.

RTL and Theme Handling: The design tokens approach allows the **same CSS variable names** to automatically reflect the active theme or direction. The React app (or Storybook) includes a **ThemeProvider** that sets data attributes on the HTML root for the selected theme and text direction [18]. For example, `<html data-theme="greenpan" dir="rtl">` for GreenPan in Arabic. CSS generated from tokens uses these attributes to scope overrides. For instance, the base `:root` might define `--t-color-bg-fill-brand` for the default Tagaddod color, and a `[data-theme="greenpan"] { --t-color-bg-fill-

`brand: ... }` block provides the GreenPan value [19]. Similarly, any RTL-specific token adjustments would be under `[dir="rtl"] { ... }` selectors [20]. This architecture means that switching the `data-theme` or `dir` attribute at runtime instantly updates all component styles (no need for JS recomputation) [21]. The ThemeProvider component manages these attributes and can persist the user's preference (e.g., storing the last chosen theme/direction in local storage) [22] [23].

Summary: In short, the **design tokens** serve as a **single source of truth** for all styling values [24]. The system is layered so that core tokens → semantic tokens → brand overrides → locale overrides all merge to produce final CSS custom properties [25]. Thanks to CSS variables and the ThemeProvider's attributes, the design system elegantly supports **multiple brands (Tagaddod/GreenPan)** and **bi-directional layouts** with minimal code changes.

React Components Architecture and Usage

The `@tagaddod/react` package contains the UI components built in React and TypeScript. Components are organized under `packages/react/src/components`, each in its own folder with a `.tsx` file (component implementation), a `.module.css` file (styles), and a `.stories.tsx` file (Storybook documentation) [26]. For example, a `Button` component would consist of `Button.tsx`, `Button.module.css`, and `Button.stories.tsx` in a `components/Button/` directory.

Styling with CSS Modules and Tokens: Each component's styling is defined in a CSS Module which uses the design token CSS variables. In the React package's global CSS (e.g. `src/styles/main.css`), the token CSS files are imported so that all `--t-` variables are available in the app [27]. This includes the core `tokens.css` as well as brand-specific and locale-specific CSS files [28]. As a result, within a component's CSS module, one can use these variables for consistent theming. For instance, a CSS snippet for the Button might look like:

```
.button {
  padding: var(--t-space-100) var(--t-space-300);
  border-radius: var(--t-border-radius-200);
  transition: all var(--t-duration-base) var(--t-easing-in-out);
}
.button.primary {
  background-color: var(--t-color-bg-fill-brand);
  color: var(--t-color-text-on-brand);
}
```

Here, the spacing, border radius, and colors are all pulled from tokens (with `--t-` prefix) [29] [30]. Because these variables adapt based on `[data-theme]` and `[dir]`, the components automatically adjust when the theme or language direction changes.

Component Implementation: In the React code (`.tsx` files), components typically use functional React with appropriate prop interfaces. The styling classes from the CSS module are applied via a utility like `clsx` to handle conditional classes. For example, the `Button.tsx` might import its CSS module as `styles` and then do:

```
<button className={clsx(styles.button, styles[variant],
styles[size])} ...>{children}</button>
```

This approach composes the base `.button` class with variant-specific classes (like `.primary` or `.secondary`) and size classes (`.sm`, `.md`, etc.) [31]. Props are defined to allow customization (variant, size, disabled, etc.), and TypeScript interfaces ensure proper prop types [32]. All components are exported from a central `index.ts` so that consumers can import from the package easily [33].

Tailwind CSS Integration: Notably, the component library integrates with Tailwind CSS v4 to provide utility classes mapped to the design tokens [34]. The token values are injected into Tailwind as CSS variables, so developers using the library can use Tailwind utility classes that automatically use design token values. For example, `className="bg-purple-1200 p-4 rounded-lg"` would correspond to the token for purple color and the standardized spacing and border radius [35] [36]. There are also custom utility classes prefixed with `t-` for direct token usage (e.g. `bg-t-surface-background` uses the background color token) [37] [38]. This means consuming apps can either use the provided React components or just use the design tokens via Tailwind classes in their JSX. The Tailwind configuration in `@tagaddod/react` imports the token CSS and uses an `@theme` directive to map tokens to Tailwind utilities [39], ensuring consistency across utility classes.

Using the Components: To use the component library in an application, one would install the package from npm and import both the components and the styles. For example:

```
import { Button } from '@tagaddod/react';
import '@tagaddod/react/dist/styles.css'; // includes all base styles and
utilities
…
<Button variant="primary" size="md">Click me</Button>
```

As shown in the library's usage docs [40], importing the distributed CSS (which bundles Tailwind utilities and token styles) is necessary so that the design tokens and base styles apply in the consumer application. After that, the React components (like `<Button>`) will automatically use the correct styles and respond to theme or RTL context if a ThemeProvider is used. The design system also provides a `ThemeProvider` component which the consuming app can wrap around its UI to enable dynamic theme switching and RTL support at runtime (by toggling the `data-theme` and `dir` as described earlier) [18] [21].

Component Display and Documentation (Storybook)

For development and documentation purposes, the design system uses **Storybook** as the "layer" to display and test components in the browser. The `@tagaddod/storybook` package sets up Storybook as a documentation site where each component has stories demonstrating its variants and states [41] [42]. Each component's `.stories.tsx` file defines a Storybook story with various props and use-cases, which allows designers and developers to interactively preview the component.

Running `yarn storybook` in the project will start Storybook locally (the configuration is defined in `packages/storybook` with a custom preview that likely wraps components in the ThemeProvider) [43]. In Storybook, you can switch between the Tagaddod and GreenPan themes and toggle RTL vs LTR to visually verify the component design under each scenario [44] [45]. This is extremely useful for catching styling issues: the guide recommends testing theme changes and RTL mode in Storybook for every component [46] [47].

The Storybook environment serves as a **living style guide**. It loads the token CSS and the React components just like an application would [48], and provides an interface for team members (designers, PMs, devs) to explore components in isolation. The repository's CI even builds and artifacts the Storybook for sharing if needed (the CI workflow builds Storybook and uploads it as an artifact) [49], and a Chromatic integration is set up for visual regression testing on the Storybook stories [50]. In summary, Storybook is the primary way to display and verify components in the browser, ensuring the design system is well-documented and easy to review.

Guide: Publishing and Maintaining the Design System

Now that we've covered the structure, below is a step-by-step guide for a product designer (with minimal coding background) to publish the design system as an npm package and make it easy for the team to use and keep updated. This guide also covers setting up versioning, continuous delivery, and basic testing in a beginner-friendly manner.

1. Preparing the Packages for NPM Publishing

Before publishing, ensure the project is properly built and versioned:

- **Set Package Versions:** Decide on an initial version number (e.g. `1.0.0`) for the component library and tokens. Update the `version` field in each package's `package.json` if needed. In this repository, the root is private (for monorepo) but the individual packages like `@tagaddod/react` and `@tagaddod/tokens` have their own versions (currently `0.0.1` in tokens [51]). You may bump these to a stable version (e.g. `0.1.0` or `1.0.0`) before first publish.

- **NPM Account:** Ensure you have an npm account (sign up on npmjs.com if not). With a free npm account, you can publish public packages. If you prefer the package not be discoverable by everyone, you could consider using a private GitHub Packages registry, but that may require your team to authenticate. For simplicity, publishing publicly on npm (or as a private GitHub package within your org) are the typical free options – npm does not support truly "unlisted" packages on free plans (public means anyone can find it).

- **Scoped Package Name:** The packages are named with the scope `@tagaddod/`. You will need to **register the scope on npm** or use an existing scope you have access to. Alternatively, you could rename packages to unscoped names (e.g. `tagaddod-react`) for publishing if obtaining the scope is an issue. If using the scope, when publishing use `npm publish --access public` because scoped packages default to private without that flag on npm's free tier.

- **Build the Packages:** Run the build to produce the distributable files. For example, `yarn build` at the root will build all packages (tokens, react, etc.) via Turborepo [52]. Ensure the build succeeds and the `dist/` outputs are generated for each package (CSS files, JS bundles, etc. as described earlier).

2. Publishing to NPM (Step-by-Step)

With preparation done, follow these steps to publish the design system packages:

1. **Login to npm:** In your terminal, navigate to the project root and run `npm login`. Enter your npm username, password, and email when prompted. This authenticates you for publishing. (Alternatively, you can use an npm access token for authentication, especially in CI, but for a manual first-time publish, login is simplest.)

2. **Publish Design Tokens:** Change directory to the tokens package and publish it first, since other packages might depend on it:

   ```
   cd packages/tokens
   npm publish --access public
   ```

   This will publish `@tagaddod/tokens` to the npm registry (assuming the version is set and no name conflicts). The `--access public` flag ensures a scoped package is public.

3. **Publish Component Library:** Next, publish the React components package:

   ```
   cd ../react
   npm publish --access public
   ```

   This publishes `@tagaddod/react`. Because the component library likely lists `@tagaddod/tokens` as a dependency, ensure that the tokens package was published successfully first (npm will require it). Repeat for any other packages you need to publish (for example, if there is an `@tagaddod/themes` package or others that the components rely on).

4. **Verify on npm:** After publishing, go to the npm website and search for your package names (or check your npm account's packages). You should see `@tagaddod/react` (and others) listed. Verify the version and that the package is public.

Tip: The repository already includes a release script using **Changesets** which can automate versioning and publishing via CI [53]. If you prefer, you can leverage this rather than manual commands. For a first manual publish, the above steps are fine. Later, you can adopt the automated flow (see CI/CD section below).

3. Making the Design System Consumable Internally

Once published, internal team members can consume the design system in their projects:

- **For Developers (Engineers):** They can install the npm packages in their projects. For example:

  ```
  npm install @tagaddod/react @tagaddod/tokens
  ```

  (If the React package lists tokens as a dependency, installing `@tagaddod/react` might pull in tokens automatically.) After installation, developers import components and styles as shown in the usage example above. This gives them access to ready-made UI components and consistent styling tokens without copying any code. They should also wrap their app in the provided `ThemeProvider` if they want dynamic theming or RTL support.

- **For Designers/Product Managers:** Non-developers might not directly install an npm package, but you can **share a Storybook** or a documentation site with them. Since Storybook is the display layer for components, consider deploying the Storybook as a static site. For example, you could use GitHub Pages or Vercel to host the Storybook build (the CI already generates a static Storybook build artifact [49]). This way, product managers can visually browse components, check variants, and even copy snippet code for usage, all without setting up a dev environment. Internally, you might circulate a link to "Tagaddod Design System Storybook" where everyone can see the latest components in both Tagaddod and GreenPan themes.

- **Documentation:** Include a **Getting Started guide** in your repository or Wiki for developers. This should cover how to install the package, import the ThemeProvider, and use a basic component. You can take content from the README (for example, the snippet showing how to import and use the Button) [40] and ensure it's easily accessible. For product managers, provide a high-level overview and link to the Storybook for visuals.

4. Ensuring Easy Updates (Versioning and Changelogs)

To make sure users of the design system can easily receive updates, follow **semantic versioning** and maintain a changelog:

- **Semantic Versioning:** Adopt a versioning strategy: increment the **patch version** for small fixes, **minor version** for new features/components that are backward-compatible, and **major version** for breaking changes. This way, internal projects can choose to update to a new major version when ready, and patch/minor updates can be picked up more frequently. For example, going from `1.0.0` to `1.1.0` means new features added; `2.0.0` would signal breaking changes.

- **Changelog:** Keep a **CHANGELOG.md** (or use GitHub Releases notes) so team members can see what changed in each version. Every time you publish a new version, list the new components, enhancements, fixes, or breaking changes. This could be as simple as a manually edited markdown file, or you could use a tool like Changesets to auto-generate it from commit messages. The repository already hints at a Changesets setup (with a workflow that can create a "Release Pull Request" and publish to npm using an NPM token) [54]. With Changesets, you would create a markdown snippet for each change, and the CI will aggregate them into release notes automatically. This is very helpful for a low-code workflow: you'd just write human-friendly descriptions of changes, and the tooling takes care of bumping versions and updating the changelog.

- **Communication of Updates:** When a new version is released, communicate to the team (via Slack or email, for instance) that "Design System vX.Y.Z is published." Highlight any important changes. Because internal developers will have the package in their `package.json`, they can update to the new version with a simple `npm update` or changing the version range. If you use a consistent naming and tagging scheme for releases (like Git tags or GitHub releases named vX.Y.Z), it will be easier to track which version is in use.

By versioning diligently and providing a changelog, **end users (developers)** can upgrade confidently, and they'll know where to look to understand new or changed behavior. This process ensures the design system evolves in a manageable way for everyone.

5. Automating with CI/CD (Continuous Integration/Delivery)

To reduce manual work and errors, you can set up a **CI/CD pipeline** to automate building, testing, and publishing the design system. GitHub Actions is a good choice, as it's integrated with your repository:

- **Continuous Integration (CI):** The repository already has a CI workflow that installs dependencies, builds all packages, and (if there were tests) runs them on each push to main [55]. Ensure this remains up-to-date. CI helps catch any errors early – for example, if a component fails to build or a Storybook story breaks, you'll know before publishing.

- **Automated Publishing:** With CI in place, you can add a step to publish to npm when a release is ready. Using GitHub Actions, you can trigger a publish on certain events (like pushing a git tag or merging a pull request with a new version). Since you may not want every commit to publish, a common approach is to use **Changesets**: you accumulate change descriptions, and when you decide to release, you run a command (or let an action run it) that bumps versions and publishes. The configuration in this repo suggests using Changesets – for instance, the workflow is set to use `changesets/action` with an `NPM_TOKEN` for authentication to npm [54]. This means once configured, simply merging a changeset that updates versions will trigger the action to publish the new package version to npm automatically.

- **Setting up GitHub Actions:** If not already set, you would need to add your **NPM_TOKEN** (an npm personal access token) to the repository's secrets. This token allows the action to publish to npm on your behalf. The CI workflow's release job already includes permission to write to packages and uses this token [56]. In practical terms, after you merge a changeset or push a version bump, the action will run `yarn release` (which calls changeset version + publish as configured) [53] and publish the packages. This automation means you, as a product designer, don't have to run `npm publish` locally each time – you would just approve a release change, and CI/CD does the heavy lifting.

- **Manual Publishing Option:** If CI/CD is intimidating at first, you can start by publishing manually (as described in step 2) and gradually introduce automation. For instance, you might first use CI to run tests and builds, but do the `npm publish` yourself for control. As you get comfortable, you can transition to a fully automated publish when a new version is tagged.

In summary, a simple CI/CD setup with GitHub Actions can watch your main branch for release triggers and then handle testing, building, and publishing to npm. This ensures consistency (every build is done the same way) and frees you from remembering all the publish steps each time. It's a bit of upfront effort, but once in place, it's very **beginner-friendly** – you just push changes and let the pipeline handle the rest.

6. Testing: Simple Setup for a Design System

Testing is important to ensure components work as expected and future changes don't break existing functionality or design. Since you may not be deeply familiar with testing frameworks, here's a simple approach:

- **Unit Tests with React Testing Library:** You can use **Jest** (a popular testing framework) along with **React Testing Library** to write basic tests for your components. This doesn't require a complex setup – you can install Jest and @testing-library/react as dev dependencies and add a `jest.config.js` (or use `vitest` if you prefer a Vite-integrated solution). For example, you might write a test for the Button that simply renders it and checks that it displays the given text, or that a disabled prop actually disables the button.

- **Snapshot Testing:** For UI components, a quick way to catch changes is snapshot testing. Jest can take a "snapshot" of the rendered output of a component and save it. If you change something later (intentionally or accidentally), the snapshot test will alert you that the output changed. This is an easy safety net for a design system – you'll know if, say, a component's DOM structure or class names changed unexpectedly. Just be cautious to review snapshots manually; updating them blindly defeats their purpose.

- **Storybook for Manual Testing:** The existing practice in the project is to use Storybook stories as a form of testing – making sure all states are covered and visually inspected [57]. Continue creating comprehensive stories (for all variants, themes, RTL, etc.) as this is essentially a visual test suite. You can even leverage Storybook's test runner or Chromatic (which the CI uses for visual regression) to catch visual differences automatically. The CI workflow has a job to publish Storybook to Chromatic for visual tests [58], which is a more advanced way to ensure UI consistency over time.

- **Integrate with CI:** Once you add some Jest tests, tie them into the CI pipeline. The current CI already calls `yarn test` (with a note that tests aren't implemented yet) [59]. You can remove the `continue-on-error: true` once you have real tests, so that failing tests will block a bad change from being released. This means every push or pull request will run the tests and alert you to any breakage, giving you confidence in merging changes.

Beginner-Friendly Tip: Start with one or two simple tests. For example, write a test for a typography component to ensure it renders the correct HTML element, or a test for a utility function if you have any in the library. You don't need 100% coverage right away. The goal is to get comfortable with the testing process. Over time, you or your team can add more tests for critical components. The key is that even a basic testing setup will significantly improve the reliability of the design system, and modern tools like React Testing Library are designed to be straightforward for beginners. There are many resources and examples online for testing React components – you can often start by copying an example and adjusting it to your components.

By following this guide, a product designer with minimal coding experience can confidently publish and maintain the Tagaddod Design System. In summary: **structure your tokens and components well, publish the library to npm for easy reuse, document everything (via Storybook and changelogs), and automate what you can (CI/CD and tests) to catch issues early.** With these practices, your internal team will find the design system easy to adopt and keep up-to-date, and you'll ensure consistency and quality across all your brand's products.

Sources:

- Tagaddod Design System Repository – README and Architecture Docs [6] [16] [19] [29] [30] [40]
- Tagaddod Design System CI Config – publishing and testing setup [53] [59] [54]

---

[1] [2] [3] [4] [5] [6] [52] README.md
https://github.com/ahmedamr-r/tagaddod-design-system/blob/f18631d32f52a20149c36043ead227cd026552d0/README.md

[7] [26] [33] [41] [42] [43] [44] [45] [46] [47] [57] Tagaddod-Design-System-Component-Development-Guide.md
https://github.com/ahmedamr-r/tagaddod-design-system/blob/f18631d32f52a20149c36043ead227cd026552d0/.claude/Tagaddod-Design-System-Component-Development-Guide.md

[8] [9] [10] [11] [12] [13] [14] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [27] [28] [29] [30] [31] [32] [48] token-components-system.md
https://github.com/ahmedamr-r/tagaddod-design-system/blob/f18631d32f52a20149c36043ead227cd026552d0/.claude/token-components-system.md

[15] complete-token-flow.md
https://github.com/ahmedamr-r/tagaddod-design-system/blob/f18631d32f52a20149c36043ead227cd026552d0/docs/complete-token-flow.md

[34] [35] [36] [37] [38] [39] [40] README.md
https://github.com/ahmedamr-r/tagaddod-design-system/blob/f18631d32f52a20149c36043ead227cd026552d0/packages/react/README.md

[49] [50] [54] [55] [56] [58] [59] ci.yml
https://github.com/ahmedamr-r/tagaddod-design-system/blob/f18631d32f52a20149c36043ead227cd026552d0/.github/workflows/ci.yml

[51] package.json
https://github.com/ahmedamr-r/tagaddod-design-system/blob/f18631d32f52a20149c36043ead227cd026552d0/packages/tokens/package.json

[53] package.json
https://github.com/ahmedamr-r/tagaddod-design-system/blob/f18631d32f52a20149c36043ead227cd026552d0/package.json