ChatGPT

Frappe Framework v15 UI Development Guide

Overview of Frappe v15 UI Architecture

Frappe v15 is a full-stack framework combining a Python backend with a rich client-side library. It provides two primary interfaces for your apps: the **Desk** (logged-in backend UI) and the **Website/Portal** (public-facing UI). In Frappe, your app's assets (JS, CSS, etc.) live in the app's `public` folder and are processed by **Bench**. The `bench build` command compiles and bundles all app assets (JS/CSS) for production, making them smaller and faster to load [1]. Frappe v15 uses a modern bundler (esbuild) that can transpile JavaScript/TypeScript, Vue Single File Components, etc., without you manually setting up Webpack or Vite. Key points to know:

- **Asset file structure:** By convention, static assets in a Frappe app go under `{your_app}/public`. Bench will symlink this to the `sites/assets/{your_app}` directory so they're served as static files.
- **Automatic bundling:** Frappe's build system automatically picks up certain files for bundling. Notably, any file with a name ending in `.bundle.js` inside your app's `public` folder is treated as an entry point and will be transpiled+bundled (including any imports) into a minified asset [2]. This means you can write modern JS (ES6+), JSX/TSX, and even import Vue/React components, and bench will bundle them for you. Frappe v15 includes **Vue 3** as a dependency out of the box, and you can also use **React or other frameworks** by adding them to your app's `package.json` and importing them in a `.bundle.js` file.
- **Desk vs Web UI:** Desk pages (accessed via `/app/<page_name>`) require user login and use the Frappe desk environment (with the sidebar, toolbar, etc.). Web/Portal pages (accessed via website routes) can be made public for non-logged-in users. The development approach differs slightly: desk pages typically use the **Page** doctype and Frappe's JS API, whereas web pages can use Jinja templates or Vue/React SPAs served through the web router.

Creating Custom Desk Pages (Page Doctype)

For internal (Desk) interfaces, Frappe provides the **Page** doctype to create full-page views in your app (not tied to a specific DocType). You can create a new Page from the Desk (go to **Pages list** at `/app/page` and add a new record) or via bench. When you create a Page, Frappe will scaffold files in your app: a folder `{module_name}/page/{page_name}` containing `page_name.json` and `page_name.js` (and an `__init__.py`) [3] [4]. The JSON file holds meta info (like page title, if it's standard), and the JS file is the client script that runs when the page is loaded.

In the generated JS (`{page_name}.js`), you'll see a template like:

```
frappe.pages['<page-name>'].on_page_load = function(wrapper) {
  const page = frappe.ui.make_app_page({
    parent: wrapper,
```

```
    title: 'None',
    single_column: true
  });
};
```

This uses `frappe.ui.make_app_page` to create a basic page container with a title [5]. The `wrapper` is the DOM element for your page content area. At this point, if you navigate to `/app/<page-name>`, you'd see an empty page with the title. You can enhance this by adding buttons, sections, or custom content either procedurally via jQuery/DOM manipulation or by mounting a front-end app (Vue/React) into the page, as described next.

Adding Actions and Content

You can use Frappe's JS API to add UI elements on a desk page. For example:

- **Actions/Buttons:** The `page` object returned by `make_app_page` has methods like `set_primary_action(label, handler)` and `add_action_item(label, handler)` to add page buttons. You can also manipulate the page toolbar, menu, etc., as needed [6] [7].
- **Content area:** Desk pages by default have a layout with a `.layout-main-section` div where you can inject content. You can target it via: `$(wrapper).find('.layout-main-section')`. For simple pages, you might directly append HTML here or use jQuery to construct elements. For complex UIs, a better approach is to mount a Vue or React app into this container (covered below).

Using Vue 3 in Custom Desk Pages

Good news: Frappe v15 natively supports Vue 3 for building reactive UIs on desk pages. You don't need to set up a separate build tooling – simply follow Frappe's conventions and let bench handle it. Here's a step-by-step guide:

1. **Create a Vue component and bundle entry:** In your app's `public/js` folder, create a subfolder (for organizational clarity, e.g. named after your page). Inside it, add a `.vue` file for your component and a `.bundle.js` file as the entry point. For example, if your page is `instrument_dashboard`, you might create:

   ```
   your_app/public/js/instrument_dashboard/InstrumentDashboard.vue
   your_app/public/js/instrument_dashboard/instrument_dashboard.bundle.js
   ```

   In the `.vue` file, you can write a standard Vue single-file component. For instance:

   ```
   <!-- InstrumentDashboard.vue -->
   <script setup>
   import { ref } from 'vue';
   const message = ref('Hello, World');
   </script>
   <template>
   ```

```
  <h1>{{ message }}</h1>
</template>
```

This is a simple example with a reactive message [8], but you can build a full component with data, methods, computed, etc. Ensure you use `<script setup>` or the Vue 3 composition API as needed (Frappe v15 comes with Vue 3).

In the `.bundle.js` file, import your Vue component and initialize a Vue app. For example:

```
import { createApp } from 'vue';
import InstrumentDashboard from './InstrumentDashboard.vue';
// function to mount the Vue app onto a given DOM element
function setup_instrument_dashboard(mountPoint) {
    const app = createApp(InstrumentDashboard);
    app.mount(mountPoint);  // mount on provided DOM element
    return app;
}
frappe.provide('frappe.ui');              // ensure namespace exists
frappe.ui.setup_instrument_dashboard = setup_instrument_dashboard;
export default setup_instrument_dashboard;
```

Here we define a global function to create and mount the Vue app. We attach it to `frappe.ui` (so it's accessible after the bundle loads) [9] [10]. You could also export a default, but attaching to `frappe.ui` or `window` is useful so you can call it from your page script.

1. **Load the bundle in your page JS:** Open the `{page_name}.js` (the one under your module's `page/` folder that was created by the Page doctype). We will modify it to load the compiled Vue bundle when the page is accessed. For example:

```
frappe.pages['instrument-dashboard'].on_page_load = function(wrapper) {
  let page = frappe.ui.make_app_page({
    parent: wrapper,
    title: 'Instrument Dashboard',
    single_column: true
  });
  // (Optional) in developer mode, hot-reload the Vue app on page loads:
  if (frappe.boot.developer_mode) {
    frappe.hot_update = frappe.hot_update || [];
    frappe.hot_update.push(() => load_vue_app());
  }
};
frappe.pages['instrument-dashboard'].on_page_show = function(wrapper) {
  load_vue_app();
};
async function load_vue_app() {
```

```
  const $container = $(wrapper).find('.layout-main-section');
  $container.empty();
  // Dynamically import the Vue bundle (ensures it's loaded from assets)
  await frappe.require('instrument_dashboard.bundle.js');
  // Now call our setup function to mount the Vue app
  frappe.ui.setup_instrument_dashboard($container);
}
```

The above code does a few things: it uses `frappe.require()` to fetch the compiled bundle file, which Frappe will serve as `/assets/your_app/js/instrument_dashboard.bundle.js`. Once the bundle is loaded, it calls our `setup_instrument_dashboard` function, passing in the jQuery-wrapped container DOM. This mounts the Vue app into the page [11]. We hook this on both page load and page show, so if the user navigates away and back (or in dev mode with hot-reload), the Vue component re-mounts. After adding this, **reload your site and navigate to the page** – you should see your Vue component's content appear inside the Frappe desk! If everything is set up correctly, for example, you'd see the "Hello, World" message from our sample component [12].

2. **Bench build & hot reload:** To compile the new assets, run `bench build`. This will transpile your `.vue` file and bundle it as specified. (Under the hood, Frappe's build will detect `instrument_dashboard.bundle.js`, use esbuild to bundle it along with its imports, including the Vue SFC, into an optimized JS file.) [13]. On successful build, refresh your page to see the result. During active development, you can use `bench watch` (or `npm run build -- --apps your_app --watch` from the `frappe` app directory) to auto-rebuild assets on file changes [14] – this significantly speeds up the feedback loop by hot-reloading the browser when you edit your Vue files.

Tip: Frappe v15 already includes Vue 3 and related tooling, so you typically do **not** need to manually add Vue to your app's dependencies. If you use additional libraries (e.g., Vue Router, Pinia, or any NPM packages), you should add them to your app's `package.json` and install via `yarn` / `npm` in your bench. Bench will then include them during the build. For example, Frappe's own form builder page uses Vue + Pinia by simply importing them in a bundle file [15] [16]. The bench build process will pick up those imports and include them in the final asset.

Using React or Other Front-end Frameworks

While Vue 3 is the "first-class" front-end in Frappe v15, you can also integrate React or others (Angular, Svelte, etc.) in your custom app. The general pattern is: create a bundle entry file that imports your framework code and attaches a mount function or component to the global scope, then use `frappe.require()` in a page or include it in a web page. Here's how you can use React as an example:

- **Set up React dependencies:** In your app folder, run `yarn add react react-dom` (this will update `package.json` with React). Since Frappe's bundler uses esbuild, it can handle JSX/TSX, but you may need to ensure appropriate file extensions. A common approach is to use a `.jsx` or `.tsx` entry file ending with `.bundle.js(x)`. For example, create `your_app/public/js/my_page/my_page.bundle.jsx`. In it, you might do:

```
import * as React from "react";
import { createRoot } from "react-dom/client";
// Define a simple React component (could also import from other files)
function HelloApp() {
  return <h1>Hello from React!</h1>;
}
// Expose a mount function
window.renderHelloApp = function(container) {
  const root = createRoot(container);
  root.render(<HelloApp />);
};
```

Here we used the React 18+ `createRoot` API to mount a component into a given DOM container. We attached the mount function to `window` (or `frappe` namespace) so it's accessible after loading. The key is naming the file with `.bundle.js` (or `.bundle.jsx`) – Frappe looks for files matching `*.bundle.js` in your app's public folder to bundle [2]. (If you use `.jsx` extension, ensure the name still ends in `.bundle.js`; e.g. `my_page.bundle.jsx` should produce `my_page.bundle.js` in assets.)

- **Load React bundle in a Desk Page:** Similar to Vue, use `frappe.require` in the page's JS. Suppose you created a Frappe Page called "My Page" (route `/app/my-page`). In `my_page.js` (under your module's folder), you can do:

```
frappe.pages['my-page'].on_page_load = function(wrapper) {
  const page = frappe.ui.make_app_page({
    parent: wrapper, title: 'My React Page', single_column: true
  });
  // Create a container div for React
  let $container = $('<div class="my-react-container"></
div>').appendTo(page.body);
  // Load the React bundle and then mount the component
  frappe.require('my_page.bundle.js').then(() => {
    // Assuming our bundle attached window.renderHelloApp
    window.renderHelloApp($container[0]);
  });
};
```

This will dynamically load `my_page.bundle.js` (which contains React and your component code), then call the mount function to render the React app into the page [17]. After a `bench build`, you should see "Hello from React!" on that page.

- **Using React in a Web Page:** For public-facing pages, you can't use `frappe.pages[...]` since that's for desk. Instead, you can include the bundle in an HTML template. One simple method is to create a static HTML file under your app's `www` folder. For example, `your_app/www/react_demo.html`:

```
<div id="react-root">Loading...</div>
{{ include_script("your_app.bundle.js") }}
<script>
  // Wait for bundle to load, then mount
  if (window.renderHelloApp) {
    window.renderHelloApp(document.getElementById('react-root'));
  }
</script>
```

The `{{ include_script(...) }}` is a Jinja directive to include your asset file with the proper path/version. After running `bench build` and refreshing, navigating to `/react_demo` on your site will serve this page and execute the script to mount the React component (showing "Hello from React"). This approach was demonstrated by community members to run React on portal pages [18]. You could alternatively use a Jinja template and attach the script via a context, but a static file is straightforward for standalone pages.

General Notes for Other Frameworks: You can integrate *any* front-end library by a similar pattern: either bundle it via the Frappe asset pipeline or include pre-built files. For example, one could build an Angular or Svelte app separately and then drop the compiled JS/CSS into `public` and reference them in `build.json` or via `include_script`. In fact, one Frappe developer used an Angular-built Web Component and included its JS/CSS in Frappe's build for use in pages [19]. The simplest route, however, is to leverage Frappe's built-in bundler: write your front-end code in modern ES6+/TS, use imports for libraries, and let `bench build` handle transpilation. Just remember these **"rules"**: - Place source JS/TS files in your app's `public` directory. - Name your entry file as `something.bundle.js` (the name before `.bundle.js` can be anything, often matching your feature/page) [2]. - If using JSX or TypeScript, you can name the file `.bundle.jsx` / `.bundle.tsx` – just ensure it ends in `.bundle.js` for the final output. (Frappe's build will output a `.js` file; the source extension can be .jsx/.tsx as needed.) - Include any needed npm packages in `package.json` and install them. Bench will detect imports (e.g. `import React from 'react'`) and bundle those packages automatically (make sure to run `bench setup requirements` or manually `yarn` to install dependencies after editing package.json). - Access your bundle in desk pages via `frappe.require('your_bundle_file.js')`, and in web pages via `{{ include_script('your_bundle_file.js') }}` or by adding to the page's HTML template. - **Don't forget to run** `bench build` whenever you add or change these files (or use `bench watch` during development).

Bench Build and Asset Compilation Details

When you run `bench build`, it processes assets for **all apps** in your bench, not just your app [20]. It will compile SCSS to CSS, concatenate/minify JS specified in build manifests, and run esbuild for any `.bundle.js` files. If one app's assets have an error, it may cause the whole build to fail or that app's assets to be skipped – so if you encounter issues (like a blank page or console errors in Frappe UI), check your build output. Common issues include syntax errors in your JS/JSX, missing imports, or misnamed files. You can also build a single app at a time with `bench build --app your_app` to isolate problems.

During development, the `bench watch` (or `npm run build -- --apps your_app --watch`) command is invaluable [21]. It will start a watch process that rebuilds assets on the fly when you edit files. In developer mode, Frappe supports **hot module replacement (HMR)** for desk pages. As shown in the Vue example, enabling `frappe.hot_update` allows your Vue app to reload without a full page refresh when the code updates [22]. This can greatly speed up UI iteration.

If you see runtime errors or crashes in the browser (e.g. a traceback pointing to Frappe but triggered by your custom code), use the browser developer console to inspect errors. For instance, an error like "Cannot find module …" or "<X> is not defined" might indicate your bundle didn't load or a global wasn't set. Ensure that the `frappe.require(...)` path matches your output asset name and that you attached any needed object to `window` or `frappe` before using it. Also, calling `frappe.require` **after** the container is in DOM (as we did in `on_page_show`) is important – if you call it too early (before the page DOM exists), your mount may fail. Following the patterns given above should avoid most of these issues.

Building Web/Portal Pages for External Users

In many apps (like your repair portal example), you'll want pages that non-logged-in users can access – for example, a page where a customer can check their instrument's repair status or where anyone can submit a repair request. Frappe offers a few ways to create web pages:

- **Web Page Doctype:** This is a doctype to create pages (usually static content or Jinja templates) from within Frappe. It's stored in the database and can use standard templating, but it's not ideal for heavy custom JS apps.
- **Static Pages in `/www`:** You can add HTML files under your app's `your_app/www` folder. Any file here becomes a page on your site (e.g. `your_app/www/status.html` is served at `yoursite.com/status`). These can include server-side templating (Jinja) and are convenient for custom code. As shown above, you can include scripts using `{{ include_script('path/to/file.js') }}` which pulls from your app's public assets. You can also include CSS with `{{ include_style('...') }}`. This method is great for embedding a Vue/React app for portal users – you basically build a small front-end and drop it into an HTML shell.
- **Web Templates + Jinja:** For more dynamic content, you can create Jinja templates in `your_app/www` or use the **Website Generator** capabilities (if you want pages tied to DocTypes). These can render context variables and so on, but if your UI is largely front-end app, you might not need heavy server templating – you can use Frappe's REST API (`/api/method/...` or client library) to fetch data via AJAX in your front-end app.

Using Your Front-end App on a Web Page: The process is similar to the static HTML example. Suppose you want a **Customer Instrument Dashboard** page on the website for logged-in customers or even anonymous users (if you allow). You could create `instrument_dashboard.html` under `www` with a basic structure (heading, a `<div id="app">` placeholder, etc.), then include your compiled bundle and call a mount. For a Vue app, you might do:

```
<h1>Instrument Tracker</h1>
<div id="app"></div>
{{ include_script('instrument_dashboard.bundle.js') }}
<script>
```

```
  // assuming your Vue bundle exposed a setup function:
  if (frappe.ui.setup_instrument_dashboard) {
    frappe.ui.setup_instrument_dashboard(document.getElementById('app'));
  }
</script>
```

If your script is written to mount automatically, you might just import it. Alternatively, you can attach your front-end as a single-page application route (see below).

Permissions: To allow non-logged-in users to access, ensure the page is not restricted. Static files in `/www` are publicly accessible by default. If you need to call server APIs as guest, you may need to allow guest access on those methods or use API tokens. Frappe also allows building **Web Forms** or **Portal Pages** (for logged in "website users"), but those are typically server-rendered forms; a custom front-end app can give more flexibility.

Advanced: Single Page Apps with Frappe (Vue/React)

For large front-end applications, managing them as bundles in Desk pages can get unwieldy. An alternative approach introduced in Frappe's ecosystem is using the **Doppio** tool (by Hussain Nagaria) to scaffold a dedicated front-end project within your app. This essentially sets up a Vite-powered Vue or React app that runs as an SPA, while still communicating with Frappe's backend. The `bench add-spa` command (added by the Doppio app) will create a new directory in your app (e.g. `/dashboard`) with a full Vue3 or React codebase (including a dev server, router, etc.) [23] [24]. It also updates your app's hooks (like `website_route_rules`) so that when you build and deploy, the SPA is served at a specific route (for example, your app's "Dashboard" SPA could be served at `yoursite.com/dashboard` as a single-page app). In development, you'd run `yarn dev` in that directory to start a Vite dev server (commonly on port 8080, proxying API calls to Frappe) [25].

The Doppio approach is great if you want a full-fledged front-end (with modern tooling, module bundling, hot reload, etc.) separate from the Frappe desk but still integrated. It also provides an option to include **Frappe UI** – a set of ready-made Vue3 + Tailwind components (available via the `frappe-ui` NPM package) for building UIs that match Frappe's style. For example, you can get a pre-built dashboard template using Vue3/Tailwind by running `bench add-frappe-ui` as mentioned in the forums [26]. This isn't mandatory – it's an accelerator if you prefer not to design components from scratch.

When to use Desk Page vs SPA: If your goal is to extend the Frappe desk (for internal users) with some custom screens (like an enhanced customer repair view for your staff), using the Page + Vue approach is ideal – it keeps your app within the Frappe UI and you can mix Frappe's UI components (dialogs, forms, etc.) with your Vue code [27] [28]. If instead you want an **external portal** that anyone can use (without logging into Frappe), and it has complex interactivity, you might lean towards an SPA served through the website. It's perfectly fine to use both in the same app: for example, your **repair_portal** app could have desk pages for internal management (repair technicians using Frappe desk) and a separate SPA or set of web pages for customers (to request service, view status, etc.). Both can reuse the same backend models and APIs.

Summary & Best Practices

Developing UIs in Frappe v15 gives you "the best of both worlds" – you can leverage modern front-end frameworks **and** Frappe's powerful backend and UI APIs together [29]. To recap the essential rules and tips:

- **File Placement:** Put your front-end source files in your app's `public` folder (e.g. `public/js/...`). For desk pages, also create the Page via the doctype so you get the `page_name.js/json` files in `your_app/{module}/page/...` [3].
- **Naming Conventions:** Name your bundle entry files with the suffix `.bundle.js` (before the extension, e.g. `myfeature.bundle.js`). The build will detect these and bundle them (including .vue, .jsx, .ts imports, etc.) [2]. In older versions one had to use `build.json`, but in v15 the `.bundle.js` naming is the easier route for custom code.
- **Including Assets:** Use `frappe.require('<bundle_name>.js')` in desk context to asynchronously load your script when needed [17]. For web context, use `{{ include_script('...') }}` in the Jinja/HTML, which handles the correct path and cache busting.
- **Bench Build:** Always run `bench build` after adding new front-end files or dependencies. This compiles your assets and will report errors if something is wrong. You can target a specific app with `--app` flag if needed. In development, run the watch/build in dev mode for rapid feedback [20] [14].
- **Dependencies:** Manage JS dependencies via `package.json`. If you add a library (like React, Chart.js, etc.), add it to `package.json` and run `yarn` (in `bench/apps/your_app`). The bundler will then allow you to `import` it in your code. (For example, to use Frappe Charts in a React/Vue page, install `frappe-charts` from NPM and import it rather than using a `<script>` tag – this avoids global namespace issues [30] [31].)
- **Frappe UI & Components:** Consider using Frappe's UI components (if using Vue3) via the `frappe-ui` library, which provides ready Tailwind-styled components. This can make your app's look-and-feel consistent with Frappe and save time on styling.
- **Testing and Debugging:** If a page crashes or is blank, check the browser console and the network tab to see if your bundle loaded. A 404 on your bundle means either the path is wrong or bench build didn't output it – check the naming/location. A runtime error like "module not found" means an import failed; ensure you installed that package or the import path is correct.
- **Cleanup on Page Hide:** When using frameworks on desk pages, it's wise to clean up when the page is unloaded. For example, if you mounted a Vue or React app, there's typically no memory leak if you replace the DOM on next load, but you might provide a hook to destroy/unmount if needed (as seen in some Frappe code, they define a `destroy()` in the class to unmount React roots when navigating away [32]). This is more of an edge case, but good to be aware of if your app holds a lot of state.
- **Leverage Frappe APIs:** Inside your Vue/React code, you can call Frappe API methods (via `frappe.call` or REST endpoints). Frappe also exposes a global `frappe` object to interact with documents, show messages, etc. For example, you can use `frappe.show_alert`, `frappe.msgprint`, or call `frappe.call({ method: ..., args: ... })` from your front-end to perform server actions. This means your modern UI can still use all the backend power of Frappe easily [33].

With this comprehensive guide, you should have a clear path to build rich UI pages in Frappe v15 – whether it's embedding a Vue3 component in a desk page or spinning up a full React portal. Follow the conventions (they're important in Frappe's "convention over configuration" philosophy), and you'll avoid the common

pitfalls that cause build or runtime issues. Happy coding, and enjoy the “low code” awesomeness of Frappe with the flexibility of modern front-end development! [34] [35]

Sources: Frappe Framework documentation and community examples for building Vue/React apps on Frappe v15 [4] [17] [18] [2].

---

[1] Bench build command technical flow - Frappe Framework - Frappe Forum
https://discuss.frappe.io/t/bench-build-command-technical-flow/132254

[2] [17] [18] [19] How to use React front-end for Page in ERPNext - App Development - Frappe Forum
https://discuss.frappe.io/t/how-to-use-react-front-end-for-page-in-erpnext/77136

[3] [4] [5] [8] [9] [11] [12] [13] [14] [20] [21] [22] [27] [28] [29] [33] Using Vue in a Desk Page
https://docs.frappe.io/framework/using-vue-inside-a-desk-page

[6] [7] [32] openwind.bundle.jsx
https://github.com/ArtisanClarinets/clarinet_lab/blob/9add558712bb328313176880002be7f7f627db6a/clarinet_lab/public/js/openwind/openwind.bundle.jsx

[10] [15] [16] form_builder.bundle.js
https://github.com/frappe/frappe/blob/395af8aa04aba0b68564f5ee8530f8de1833ea09/frappe/public/js/form_builder/form_builder.bundle.js

[23] [24] [25] GitHub - NagariaHussain/doppio: A Frappe app (CLI) to magically setup single page applications and Vue/React powered desk pages on your custom Frappe apps.
https://github.com/NagariaHussain/doppio

[26] [30] [31] Vue js / React installation in Frappe - Frappe Framework - Frappe Forum
https://discuss.frappe.io/t/vue-js-react-installation-in-frappe/105933

[34] [35] Version 15
https://frappeframework.com/version-15