ChatGPT

Rapidly Reconfigurable Platform Monorepo Architecture

Overview: This document presents a reference scaffold for a **monorepo** architecture that supports rapid reconfiguration through a **capability-based plugin model**. We outline the repository layout, key architecture components for both frontend and backend, and specifications for adding new **capability plugins**. The goal is a modular yet cohesive system where new features (plugins) can be added or removed with minimal impact on the core, enabling extensibility, flexibility, and isolated development [1] [2]. We also address cross-cutting concerns like multi-tenancy, feature flags, and RBAC, and emphasize a developer-friendly setup with consistent scaffolding and documentation.

Monorepo Structure and Shared Components

The project is organized as a single monorepository containing multiple apps and packages. This allows code reuse via shared libraries while keeping modules isolated [2]. A high-level layout is as follows:

```
project-root/
├── apps/
│   ├── frontend/                # React app (App Shell)
│   │   ├── src/
│   │   │   ├── App.tsx          # Core AppShell, routing, plugin loader
│   │   │   ├── components/      # Core layout components, navigation
│   │   │   └── plugins-manifest.ts  # Plugin manifest loader/registry
│   │   ├── public/              # Static assets
│   │   └── package.json         # Vite + React config
│   └── backend/                 # FastAPI app
│       ├── app/                 # Python application package
│       │   ├── main.py          # FastAPI initialization, plugin registration
│       │   ├── api/             # Core API routes (auth, health, etc.)
│       │   ├── plugins/         # Backend plugin modules (capabilities)
│       │   │   ├── __init__.py
│       │   │   ├── dropbox_plugin/   # Example plugin module (Dropbox)
│       │   │   └── ...other plugins...
│       │   ├── core/            # Core services (auth, DB, config, RBAC)
│       │   ├── models.py        # Shared DB models (common schema)
│       │   └── config.py        # App configuration (including feature flags,
multi-tenancy)
│       ├── requirements.txt
│       └── pyproject.toml
├── packages/
│   ├── ui-components/           # Shared React component library (design
system)
│   │   ├── src/ (Button.tsx, Modal.tsx, etc.)
│   │   ├── src/tokens/          # Design tokens (colors, spacing, etc.)
```

```
│   │   ├── src/index.ts        # Entry point exports all components
│   │   ├── package.json
│   │   └── .storybook/         # Storybook config for component library
│   ├── sdk/                    # Shared SDK (TypeScript utilities for API
calls, types)
│   ├── integrations/           # Integration-specific packages
│   │   └── dropbox/            # (Optional) Dropbox integration shared code
│   └── plugin-core/            # Plugin interface definitions (if separated)
├── docs/                       # Documentation, playbooks, developer guides
├── .github/                    # CI/CD workflows (tests, lint, etc.)
└── package.json                # Root for yarn/pnpm workspaces
```

Monorepo tooling: We use Yarn (or PNPM) workspaces to manage Node packages for the frontend and shared JS/TS packages. This means `apps/frontend` and `packages/*` are part of one npm workspace, enabling seamless import of the shared `ui-components` library into the React app without separate versioning [3]. The Python backend is in the same repo for consistency, though it has its own dependency management (e.g. a virtualenv or Poetry/PDM setup). CI pipelines can run tests and builds across the whole monorepo and coordinate releases.

Shared UI library: The `packages/ui-components` directory contains a **design system** implemented as a reusable React component library (around 15–25 components). It includes foundational components like `<Button>`, `<Modal>`, `<FormField>`, etc., all following consistent design guidelines. Design tokens (colors, typography scales, spacing units, etc.) are defined in `src/tokens/` and exported (e.g. as CSS variables or via a style dictionary) to ensure a single source of truth for styling. By centralizing these, the entire platform (core app and plugins) shares a consistent look and feel. Using a monorepo for this design system means multiple apps or modules can reuse UI components easily, with isolation and without duplicating code [2].

Storybook integration: We integrate **Storybook** for the component library to serve as a **component index and usage guide**. The `.storybook` config in `ui-components` allows running Storybook to interactively showcase all reusable components with their various states. Each component has accompanying stories demonstrating usage, which act as both living documentation and a testing ground for design. Storybook serves as a **developer hub** for UI recipes – new developers or even AI-assisted developer agents can query available components and see examples of how to use them. (In the future, we can leverage tools like *Storybook MCP* which exposes component metadata to AI agents [4], making the scaffold "agent-friendly" by allowing automated tools to list components and their props via an API.)

Consistent naming and structure: The monorepo enforces conventions for naming and layout. For example, all capability plugins (both frontend and backend parts) use a consistent naming scheme (e.g. `something_plugin` for backend Python package and `somethingPlugin` or similar for frontend). Following naming patterns (as seen in frameworks like Backstage [5]) makes it easy to identify plugin-related code. Shared packages use an `@scope/name` syntax (if using a package manager scope) such as `@ourorg/ui-components` for clarity. This consistency improves developer onboarding and also helps programmatic tools or scripts to auto-discover modules.

Frontend Architecture: React App Shell & Plugins

The **frontend** is a **React** application (bootstrapped with Vite for fast builds) that implements an **App Shell** architecture. The App Shell provides the core layout and navigation, while feature modules are delivered as plugins that can be composed into the app dynamically via manifest definitions. This approach allows new UI features to be added without altering the core shell code, aligning with the plugin design pattern of extending functionality without modifying the host [1].

App Shell and Dynamic Plugin Composition

The App Shell (`App.tsx`) is the container application responsible for high-level UI structure: e.g. global navigation menus, routing container, top bar, etc. It declares **extension points** where plugins can hook into. For example, the shell might have a navigation menu component that can accept additional menu items from plugins, and a routing system (using React Router or similar) that can add new routes for plugin pages. Initially, the shell might render only core routes (like Dashboard, Settings, etc.), but it is designed to incorporate plugin-provided routes during initialization.

Plugin manifest: Each frontend plugin provides a manifest (could be a JSON file or a TypeScript module export) describing its contributions to the UI. A manifest defines things like: - **Routes** – e.g. a path and the React component to render for that path. - **Navigation entries** – e.g. a label/icon and target route to appear in the sidebar or top menu. - (Potentially other extensions, like if the plugin wants to contribute a widget to a dashboard, etc., though initially we focus on pages and nav items.)

For example, a plugin manifest might look like:

```
// apps/frontend/src/plugins-manifest.ts (aggregating all plugin definitions)
import { DropboxPage } from '@ourorg/dropbox-plugin/frontend';  //
hypothetical import

export const pluginManifests: CapabilityManifest[] = [
  {
    name: "DropboxIntegration",
    routes: [
      { path: "/integrations/dropbox", component: DropboxPage }
    ],
    navigation: [
      { label: "Dropbox", path: "/integrations/dropbox", icon: "📦" }
    ]
  },
  // ...other plugin manifests
];
```

In this TypeScript structure, we define a list of `CapabilityManifest` objects. The App Shell on startup will iterate over `pluginManifests` and register each route with the router and add each nav item to the menu. The components can be code-split (e.g. using `React.lazy` and dynamic import for the plugin component) so that the plugin code is loaded only when needed, keeping initial bundle size small. For instance, instead of directly importing `DropboxPage`, the manifest could provide a loader: `loader: () => import('@ourorg/dropbox-plugin/frontend').then(m => <m.DropboxPage/>)`. This pattern is inspired by frameworks like Backstage which use lazy-loaded extensions [6].

Dynamic enabling/disabling: Because plugins are registered via manifests, it's straightforward to enable or disable a plugin by including or removing its manifest (or by checking a feature flag before registration). The manifest approach acts as a **configuration** for the app composition [7]. We could even store plugin manifests externally or generate them based on user tenancy/config. For instance, if a certain tenant should not have the Dropbox feature, the manifest for that plugin can be filtered out at runtime for that tenant's configuration. This gives a flexible way to turn features on/off without redeploying code, honoring feature flags and multi-tenant settings (the app could load a different manifest list per tenant login or subdomain).

Routing architecture: We use React Router (v6+) for client-side routing. The App Shell defines a router that includes core routes and then maps plugin routes. For each `manifest.routes` entry, we call `router.addRoute` or include it in our route configuration. Each plugin route might be namespaced (for example, prefixed with `/plugin-name/*` or like the `/integrations/dropbox` example above). This mirrors how the backend API might also namespace plugin endpoints, maintaining a clear separation.

Navigation integration: The shell's Navigation component will read all plugin `navigation` entries and merge them into the menu. Each entry can specify a label, target route, and possibly an icon or required permission. We ensure the nav is dynamically generated from these manifests so that adding a new plugin automatically adds a menu item (if the user has access). We can also incorporate feature flag conditions or RBAC checks here – e.g., not showing the menu item if the plugin is disabled or the user lacks the role to use it.

Shared Component Library and Design Tokens

As mentioned, the frontend leverages a shared **UI component library** (`packages/ui-components`). All common UI elements are built there using TypeScript + React, and are imported into the App Shell and plugins as needed. This promotes consistency in style and behavior across all capabilities. For example, if every form uses a `<FormField>` from the library, then any change in form UX (say adding inline validation messaging) can be made in one place and applied everywhere.

Design tokens: The design tokens (e.g. color palette, font sizes, spacing scale, etc.) are defined in a format that can be shared between tools. For instance, colors might be defined in a JSON or TS file and also output as CSS custom properties. The React components consume these tokens (perhaps via a Context or CSS variables) to ensure consistent theming. The use of tokens makes it easier to implement **theming or tenant-specific branding**. If multi-tenant requirements include different branding per tenant, we could generate a different set of CSS variables for each tenant's theme, but the components remain the same, referencing the abstract tokens.

Storybook for design system: Running Storybook for `ui-components` allows developers and designers to collaboratively develop UI in isolation. Each component's story demonstrates its usage (props, states, etc.), effectively serving as **usage recipes** for that component. For example, a `Button` component story might show "Primary Button", "Disabled Button", etc., with code snippets for how to invoke them. This not only aids human developers but also can be parsed by documentation tools or AI agents to understand how to use the component. We maintain a **component index** in Storybook (which is essentially the sidebar listing all components). Combined with naming consistency and clear prop documentation, this makes the system highly **developer-friendly**. New team members can quickly find what components already exist and how to use them, reducing duplicated work and misalignment.

We enforce **naming consistency** in the component library (e.g. all components use PascalCase for React components, and their filenames match the component name). Similarly, design tokens use a consistent naming scheme (e.g. `color.primary.100`, `spacing.md`, etc.) and are documented in the Storybook or in `docs/styleguide.md`. This consistency ensures everyone (and every tool) speaks the same language when referencing styles, simplifying coordination.

Backend Architecture: FastAPI Modular Monolith with Plugins

The **backend** is implemented in Python using **FastAPI**. We adopt a **modular monolith** approach where the backend is a single deployed application, but internally structured as logically independent modules or "capabilities." Each capability is implemented as a **plugin module** that can encapsulate its own API routes, background jobs, and other logic. This gives the development-time benefits of microservices (clear boundaries, independent development/deployment of features) while keeping runtime simplicity (one process, one deployment) [8] [9].

Modular Monolith Structure

Inside `apps/backend/app`, we separate core functionality from plugin features. Core concerns (like authentication, database connection, user management, common utilities) reside in the `core/` or `api/` directories. Feature-specific logic resides in `plugins/`. Each plugin is a Python package (or module) that follows a defined interface to register itself with the main FastAPI app.

For example, `app/main.py` will do something like:

```
from fastapi import FastAPI
from . import core, plugins

app = FastAPI(title="Reconfigurable Platform API")

# Initialize core systems (DB, auth, config)
core.init_app(app)

# Discover and include plugins
for plugin in plugins.discover_enabled():
    app.include_router(plugin.router, prefix=plugin.prefix,
tags=[plugin.name])
    plugin.register_jobs(core.scheduler)
```

Here, `plugins.discover_enabled()` might scan the `app/plugins/` directory for any plugin modules that are enabled via config. This is analogous to the plugin registration concept in design pattern terms [10] – the core is finding extension modules and incorporating them. We can implement this discovery by importing all sub-packages in `app.plugins` dynamically (using `importlib`) or via a configuration list (e.g., read from `config.py` which plugins to load). The key is that adding a new plugin should **not** require manually editing a central router import; it should be auto-discovered, making it plug-and-play [11] [12].

Each plugin module in `app/plugins` is structured, for instance, as `dropbox_plugin/` containing its own `routes.py`, maybe `schemas.py` for Pydantic models, `service.py` for any logic, etc. The plugin exports an object (or uses a well-known naming convention) that the main app can pick up. For

simplicity, we might say each plugin's `__init__.py` instantiates a `router` (FastAPI `APIRouter`) and perhaps a `register_jobs` function. The `prefix` can be defined in the router itself or as a constant. In our example above, `plugin.router` would already have a prefix set (like `/integrations/dropbox`) when the router is created, or we pass the prefix in `include_router`.

Using FastAPI's router include mechanism is akin to providing **extension points** (the core app extends itself with each plugin's routes). Alternatively, FastAPI supports mounting sub-applications for complete isolation [13]. In a more advanced scenario, each plugin could be a sub-FastAPI app mounted at e.g. `/api/plugins/<name>`, which would isolate OpenAPI docs and middleware. However, mounting sub-apps can complicate things like dependency sharing (each sub-app is somewhat independent). For our purposes, using routers is sufficient and allows all plugins to share the core app context (db session, auth dependencies, etc.) easily.

Internal communications: All modules run in-process, so plugins can call core services directly (e.g. use a function from `core.user_service` to get the current user or to log an event). However, to preserve **loose coupling**, plugins are encouraged to use only the public interfaces of core modules and not reach into each other's internals. This is aligned with the idea of separation of concerns – each plugin is focused on its feature [14]. If communication between plugins is needed, they should go through well-defined interfaces or the database, rather than arbitrary imports, to keep the system maintainable.

The core system provides common infrastructure that plugins can leverage so they don't reinvent the wheel. For example, core could provide a logger, database session, and security utilities. As noted in plugin architecture principles, the core often contains shared components like logging, security, DB access so that plugins remain lean [15]. In our setup, `core/` includes things like database models and a session (perhaps using SQLModel or SQLAlchemy), and authentication (FastAPI dependency that checks JWT or session and yields a `current_user`). Plugins can import these – e.g., a plugin route can `Depends(core.auth.get_current_user)` to secure an endpoint.

API Design and Plugin Interface

We define a **capability plugin interface** on the backend to formalize how new features integrate. At minimum, each plugin must provide: - One or more **API routers** (FastAPI routers with path operations) that implement its HTTP API. - (Optionally) **Background tasks or jobs** if the feature needs periodic work or async tasks (e.g. scheduled syncs, or Celery tasks). - (Optionally) **Event handlers** for application startup/shutdown if needed (to initialize something). - Metadata like a **name** (used for tagging routes or identifying the plugin in logs).

We can express this with a base class or Protocol (using Python's typing). For example:

```
# packages/plugin-core/interfaces.py (if using a shared package for plugin
contracts)
from fastapi import APIRouter
from typing import Protocol

class BackendPlugin(Protocol):
    name: str
    router: APIRouter        # Router with all plugin endpoints
    def register_jobs(self, scheduler): ...
```

```
    def on_startup(self, app): ...
    def on_shutdown(self, app): ...
```

A plugin module would instantiate an object fulfilling this interface. For simplicity, some plugins might just define `router` and a no-op `register_jobs`. The main app's discovery will treat any module in `app.plugins` as a plugin if it defines the expected attributes (router, etc.). This is our **"point of contract"** where the plugin announces itself and its integration points to the core [7].

Example router registration: The Dropbox integration plugin might create a router like:

```
# app/plugins/dropbox_plugin/routes.py
from fastapi import APIRouter, Depends
from ...core.auth import get_current_user  # dependency from core
router = APIRouter(prefix="/integrations/dropbox", tags=["Dropbox"])

@router.post("/files")
def upload_file(file: bytes, user=Depends(get_current_user)):
    # logic to upload a file to Dropbox for this user
    ...

@router.get("/files")
def list_files(user=Depends(get_current_user)):
    # logic to list Dropbox files for this user
    ...
```

This router is then exposed via the plugin's `__init__.py` perhaps:

```
# app/plugins/dropbox_plugin/__init__.py
from .routes import router
name = "DropboxIntegration"

def register_jobs(scheduler):
    # e.g., schedule periodic sync job if needed
    # scheduler.add_job(sync_dropbox, trigger="interval", hours=1)
    pass
```

The main app will import `app.plugins.dropbox_plugin` and find `router` and `register_jobs` in it (we could also encapsulate these in a class, but using module-level variables for simplicity is fine). It then does `app.include_router(dropbox_plugin.router, prefix="/api")` (if we want a global API prefix) or directly, since we already included prefix in the router.

Background jobs: For background processing, we might integrate an async task scheduler. We could use **APScheduler** for in-process scheduling of jobs or integrate **Celery/RQ** for distributed task processing. As a foundation, our core sets up a `scheduler` (for APScheduler) or a Celery app. The plugin's `register_jobs` is given this scheduler to schedule any recurrent tasks. For example, the Dropbox plugin might schedule a daily job to refresh tokens or sync files. If Celery is used, `register_jobs` might just declare tasks (Celery will autodiscover tasks in plugin modules). In any case, the plugin interface allows registering these jobs so that the core knows about them.

Database and models: In a modular monolith, one approach is a single database schema with tables possibly namespaced by module. We can allow plugins to define their own DB models (e.g., Dropbox plugin might define a `DropboxAccount` model to store OAuth tokens per user). These can be integrated via migrations – e.g., each plugin could have a Alembic migration script that the main Alembic config includes. To keep things simple, core can contain the base models and each plugin can either register models with the core's DB or use something like SQLModel where models get added to the metadata automatically when imported. Ensuring multi-tenancy might affect the DB layer (see below).

Multi-Tenancy, Feature Flags, and RBAC Foundations

Multi-Tenant configuration: The architecture supports multi-tenancy by isolating tenant data and configuration. We assume a single deployment serves multiple tenants (for example, multiple customer organizations). There are a couple of strategies to handle this: - **Data isolation:** We include a tenant identifier in data models (e.g., each key entity has a `tenant_id` column) and scope all queries by the current tenant. The `core.auth.get_current_user` could embed the tenant context (perhaps in the user's JWT or session). We might use a dependency that provides `current_tenant` based on the request subdomain or claims. All plugin logic can then retrieve `current_tenant` and ensure they only operate on that tenant's data. For example, the Dropbox plugin when listing files might filter by the user (who is implicitly tied to a tenant). - **Config per tenant:** The `config.py` or a database config store can specify which features are enabled for which tenant. For instance, Tenant A might have Dropbox integration enabled, Tenant B not. We can implement this by having a tenant configuration dictionary that the backend checks on each request or at least when initializing a plugin. If a plugin is globally enabled but not allowed for a specific tenant, the API endpoints could return 403 or be hidden. On the frontend, the manifest loading could be tenant-aware (loading only manifests for allowed plugins for that tenant's user).

Our scaffolding includes a `config.py` that handles loading configuration from environment or files (including multi-tenant settings and feature flags). For example:

```
# app/config.py
ENABLED_PLUGINS = ["dropbox_plugin", "another_plugin"]
TENANT_FEATURES = {
    "tenantA": {"dropbox_plugin": True, "another_plugin": False},
    "tenantB": {"dropbox_plugin": False, "another_plugin": True},
}
```

The `plugins.discover_enabled()` function mentioned earlier can consult these settings to only load plugins that are globally enabled. Additionally, inside each plugin's logic, one can check `TENANT_FEATURES` for the current tenant if finer control is needed (though ideally requests for disabled features wouldn't reach the logic at all, because the frontend and maybe backend routing would skip registration).

Feature flags: Beyond tenant-level enablement, we might have feature flags for gradual rollouts or testing. For example, a plugin might be in beta and behind a flag. We can integrate a feature flag library or service; but even a simple approach is to use config values (environment variables) that are checked before registering a plugin or before executing certain code paths. In the code, this might look like:

```
if config.FEATURE_DROPBOX:
    app.include_router(dropbox_plugin.router, ...)
```

On the frontend, a corresponding check could prevent adding the Dropbox nav item if the feature is off. Our design allows these checks centrally at manifest/registration level, so turning off a feature flag effectively unhooks the plugin system-wide. More granular feature flags (like toggling a specific functionality within a plugin) can be handled within that plugin's code or via the config passed to it.

RBAC foundations: Security is critical in a multi-tenant, multi-feature system. We establish a basic **Role-Based Access Control** layer in the core. This includes: - A concept of **roles** and **permissions** in the data model (e.g., a User model in core with roles, and possibly a Permission model or simple enum). - The authentication module (e.g., OAuth2 with JWT or session auth in FastAPI) will, upon login, assign roles to the user's token. - We provide dependency utilities like `Depends(require_role("admin"))` or `Depends(require_permission("dropbox:read"))`. These can be implemented by checking `current_user.roles` and raising HTTP 403 if not authorized.

Each plugin can specify what roles or permissions are required for its endpoints. For example, the Dropbox plugin might require that the user has a permission `"integrations:dropbox"` to use it. We could enforce this by applying a dependency on the router or individual routes:

```
from ...core.auth import require_permission

@router.get("/files",
dependencies=[Depends(require_permission("integrations:dropbox"))])
def list_files(...):
    ...
```

This way, even if a plugin is enabled and loaded, not every user can use it – only those with appropriate permissions. We envision that the RBAC system is flexible: perhaps a default role like "Admin" gets all features, while "User" has only some. The foundation is there for future enhancement (like more complex policy rules or integration with a third-party authZ system), but at minimum we include role checks in our scaffold.

Importantly, the RBAC is multi-tenant aware: roles are scoped to each tenant (e.g., "Admin" in tenant A is different from "Admin" in tenant B). Our data model would reflect that (if using separate user records per tenant or a mapping table). This ensures that one tenant's admin cannot access another tenant's data, which the combination of tenant scoping and RBAC guarantees.

Logging and Monitoring: Though not explicitly asked, it's worth mentioning that with multiple modules, debugging can be aided by structured logging that includes plugin name or tenant context. We configure a logger (using Python's `logging` or structlog) to automatically include the plugin name in log entries, perhaps by configuring each router with a custom logger or using context variables. FastAPI middlewares could insert tenant ID and plugin name (if using sub-apps or route tags) into logs [16]. This is a consideration for an ops-friendly design.

Capability Plugin Interface Specification

To add a new **capability plugin** to this system, a developer should follow a defined pattern so that the plugin cleanly interfaces with both the frontend shell and backend core. Below is a specification (interface contract) for plugins:

- **Naming & Packaging:**
- Backend: Create a Python package under `app/plugins/` (e.g., `myfeature_plugin`). Give it a clear name attribute (e.g., `name = "MyFeature"`).

- Frontend: Create either a corresponding entry in the manifest or a dedicated TS module in `packages/` (e.g., a new package `packages/myfeature-plugin` for front-end components, or simply add to a manifest config if the plugin is simple). Ensure the plugin's name is consistent across front and back.

- **Backend Plugin Interface:** The plugin module must expose at least:

- `router`: a FastAPI `APIRouter` with all the plugin's API endpoints defined. Use a **prefix** that is unique to the capability (e.g., `/myfeature` or `/<domain>/myfeature`) to avoid route conflicts. Tag the routes appropriately (for documentation grouping).
- `register_jobs(scheduler)`: a function that accepts the scheduler (or any job orchestrator) and schedules any required background tasks. If the plugin has no background jobs, this can be a no-op or omitted.
- `name`: a human-readable name or identifier (could be used in logs or in assembling UI, etc.).
- Optionally, `init_app(app)` for any special integration on startup (for example, if plugin needs to register event handlers or startup logic).
- Optionally, `models.py` if the plugin has database models; these should be imported somewhere during startup so that migrations see them. (Our migration tool should be set to scan all plugin modules for models.)

- The plugin **should not start its own server** or event loop; it runs within the FastAPI app's context.

- **Frontend Plugin Interface:** The plugin should integrate with the App Shell through the manifest:

- **Route(s):** Define one or more routes that the plugin contributes, each with a path and a React component (or lazy component loader). The path should likely correspond to the backend API prefix for consistency (e.g., front-end route `/app/myfeature` might use backend APIs under `/api/myfeature`). Usually, a plugin will have at least one main page component.
- **Nav item(s):** Provide navigation config: a label, route (matching one of the routes above), and optionally an icon or position. The nav config could also include a role/permission requirement string if the item should only show for certain users (the shell can compare the current user's roles).
- **Bundling:** If using a separate package for the plugin's frontend, ensure it is added to the monorepo workspace and built/linked into the app. If using a central manifest file (like an array of manifests), ensure to import the plugin's component or module in that file so it gets included in the build.

- **State management:** If the plugin needs state, it can use the global Redux/Zustand store of the app or context. The plugin should define any special context providers in its component if

needed, but generally it should rely on core contexts for things like the current user, theme, etc., instead of duplicating them.

- **Configuration:** If the plugin requires configuration (API keys, etc.), define how this is provided:

- Backend config (e.g., in `config.py`, have entries like `DROPBOX_API_KEY` for Dropbox plugin). The plugin can fetch this from `app.core.config` at runtime.
- Frontend config (if needed, e.g., feature flag value or third-party keys for JS SDKs) could be passed via a global JS config object or .env for the front-end. Usually, sensitive keys remain in backend; front-end might only need something like a client ID or just use backend endpoints.

- Multi-tenant config: If certain settings vary per tenant (like a different Dropbox app key per tenant), the core should provide a mechanism to store and retrieve that. Possibly through a database table that the plugin can query (e.g., a table of IntegrationCredentials keyed by tenant and integration type). This is an advanced use-case but our architecture is aware of it.

- **Permissions:** Define any RBAC permissions that guard the plugin's functionality:

- Decide on a permission name or set of names (e.g., `"myfeature.read"`, `"myfeature.write"`). Document these in the plugin README or `docs/`.
- Use the core's `require_permission` or `require_role` dependencies in your router to enforce security.

- Frontend should ideally hide or disable UI for which the user lacks permission. For example, if a user doesn't have `"myfeature"` permission, the plugin's nav item might be omitted. The shell can handle this if we include a permission field in the manifest and filter accordingly.

- **Testing:** Every plugin should have its own unit tests (e.g., in `tests/test_myfeature_plugin.py`) to test its routes and logic in isolation. The core test harness can include all plugins, but plugins can also be tested with FastAPI's `TestClient` by including their router in a temporary app. This ensures that adding a new plugin doesn't break others and vice versa.

By adhering to this interface, a new capability can be developed independently and "plugged in" easily. The core system doesn't need to be aware of implementation details of plugins – it just needs to load them via the interface [17] [10]. This keeps the core clean and the plugins independent, reflecting the plugin architecture's principles of modularity and extensibility [18] [19].

Example Plugin: Dropbox Integration

To make the above concrete, consider adding a **Dropbox Integration** capability to the platform. This plugin will allow users to connect their Dropbox account and perform file operations from within our application. We'll outline how this plugin fits into the monorepo and provide sample code snippets for key parts.

1. Monorepo placement:

- **Backend:** `apps/backend/app/plugins/dropbox_plugin/` – contains `__init__.py`, `routes.py`, and any other logic (e.g. `client.py` for Dropbox API client).

- **Frontend:** We have a couple of options. For simplicity, we might not create a completely separate package, but instead put the UI for Dropbox in a shared plugins manifest. Alternatively, create `packages/integrations/dropbox/` for front-end components if it's large. Let's assume a separate package for clarity:
- `packages/integrations/dropbox/` – contains `DropboxPage.tsx` and maybe smaller components like `DropboxFileList.tsx`. It also exports a manifest entry used by the App Shell.

2. Frontend integration (DropboxPage component and manifest):

```
// packages/integrations/dropbox/DropboxPage.tsx
import React, { useEffect, useState } from 'react';
import { Button, Modal } from '@ourorg/ui-components';  // using shared
components
import { apiClient } from '@ourorg/sdk';  // hypothetical API client for
backend

export const DropboxPage: React.FC = () => {
  const [files, setFiles] = useState<string[]>([]);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    apiClient.get('/integrations/dropbox/files')
      .then(res => setFiles(res.data))
      .catch(err => setError(err.message));
  }, []);

  const handleUpload = async (file: File) => {
    // call backend to upload file (simplified)
    try {
      await apiClient.post('/integrations/dropbox/files', file);
      setFiles(prev => [...prev, file.name]);
    } catch (e:any) {
      setError(e.message);
    }
  };

  return (
    <div>
      <h1>Dropbox Integration</h1>
      {error && <div className="error">{error}</div>}
      <Button onClick={() => {/* trigger file input */}}>Upload File</Button>
      <ul>
        {files.map(name => <li key={name}>{name}</li>)}
      </ul>
      {/* Modal from shared components could be used for OAuth flow or file
picker */}
    </div>
  );
};
```

This React component uses the shared UI library (`Button`, `Modal`) and a hypothetical `apiClient` from our SDK package to communicate with the backend. It fetches a list of files from our backend API (`GET /integrations/dropbox/files`) and allows uploading a file via `POST /integrations/dropbox/files`. We manage state with React hooks.

We would add this component to the plugin manifest so the App Shell knows about it:

```
// apps/frontend/src/plugins-manifest.ts (excerpt for Dropbox)
import { DropboxPage } from '@ourorg/dropbox';  // assuming index.ts exports DropboxPage

pluginManifests.push({
  name: "DropboxIntegration",
  routes: [
    { path: "/integrations/dropbox", component: DropboxPage }
  ],
  navigation: [
    { label: "Dropbox", path: "/integrations/dropbox", icon: "📦" }
  ],
  // maybe include a permission requirement:
  requiredRole: "user",  // (or a permission like 'integrations:dropbox')
});
```

Now, when the front-end app loads, it will include a “Dropbox” item in the nav. If the user clicks it, React Router will render `<DropboxPage>` which calls the backend. If the feature flag for Dropbox is off or the user lacks permission, the nav item can be conditionally excluded (for instance, we check something like `if (!user.permissions.includes('integrations:dropbox'))` before pushing the manifest).

3. Backend integration (Dropbox plugin backend):

```
# apps/backend/app/plugins/dropbox_plugin/routes.py
from fastapi import APIRouter, Depends, HTTPException
from ...core.auth import get_current_user  # core dependency
from ...core import db  # hypothetical database session or functions

router = APIRouter(prefix="/integrations/dropbox", tags=["Dropbox"])

@router.get("/files")
def list_files(current_user=Depends(get_current_user)):
    """List files from the user's Dropbox."""
    # current_user could contain a token or ID to access Dropbox
    token = current_user.dropbox_token
    if not token:
        raise HTTPException(status_code=400, detail="Dropbox not linked")
    # Here we would call Dropbox API (using their SDK or HTTP calls).
    files = fetch_files_from_dropbox(token)  # pseudo-function
    return [f.name for f in files]
```

```
@router.post("/files")
def upload_file(file: bytes = Depends(...),
current_user=Depends(get_current_user)):
    """Upload a file to Dropbox."""
    token = current_user.dropbox_token
    if not token:
        raise HTTPException(status_code=400, detail="Dropbox not linked")
    # Save the file to Dropbox via API
    success = upload_to_dropbox(token, file)
    if not success:
        raise HTTPException(status_code=502, detail="Upload to Dropbox
failed")
    return {"status": "ok"}
```

In this simplified code, we assume `current_user` has a property `dropbox_token` which was obtained earlier (the process for linking an account would involve an OAuth flow not fully shown here – likely another endpoint to handle the OAuth callback and store the token). The plugin's endpoints use `get_current_user` to ensure the user is authenticated and to get user-specific data. We then call out to hypothetical helper functions `fetch_files_from_dropbox` and `upload_to_dropbox` which would interact with the Dropbox API. Those could be implemented using the official Dropbox Python SDK or plain HTTP calls; they might reside in a `client.py` in the plugin.

We also handle errors by throwing HTTP 400 if the user hasn't linked their Dropbox (no token) or a 502 if the Dropbox API call fails. This provides clear feedback to the frontend.

The plugin would also include a `register_jobs` if needed. Suppose we want to periodically sync some Dropbox info (though in this simple example, maybe not needed). If we did:

```
# apps/backend/app/plugins/dropbox_plugin/__init__.py
from .routes import router

name = "DropboxIntegration"

def register_jobs(scheduler):
    # e.g., a daily job to clean up any expired tokens or sync something
    # scheduler.add_job(sync_dropbox_accounts, trigger="cron", hour=3)
    pass
```

Now, to wire it up, ensure `apps/backend/app/plugins/__init__.py` has logic to discover this plugin. If using an explicit list in config:

```
# apps/backend/app/plugins/__init__.py
import importlib
from .. import config

_plugins = []
for plugin_name in config.ENABLED_PLUGINS:  # e.g., ["dropbox_plugin", ...]
    module = importlib.import_module(f"app.plugins.{plugin_name}")
```

```
    _plugins.append(module)

def discover_enabled():
    return _plugins
```

The FastAPI app in `main.py` will use this as shown earlier, including each plugin's router and calling its `register_jobs`.

4. RBAC and multi-tenant for Dropbox: In a real scenario, we'd likely want only certain roles to access Dropbox integration (maybe it's a premium feature). We could enforce that by adding `dependencies=[Depends(require_permission("integrations:dropbox"))]` on the router or individual endpoints. If a user without that permission tries to call the API, they'd get 403. On the frontend, we would ideally not show the Dropbox menu to unauthorized users in the first place. The manifest entry could have `requiredPermission: 'integrations:dropbox'` and the shell could filter it out based on the logged-in user's permissions.

For multi-tenancy: some tenants might not enable Dropbox at all. We reflect that by not including the plugin if `TENANT_FEATURES[current_tenant]['dropbox_plugin'] == False`. That could happen at login time (the backend might embed allowed features in the user's token or profile). The frontend could also receive that info (e.g., an API call on login returns enabled features for that tenant). If disabled, again, we don't show it. And even if somehow a call is made, the backend can double-check the tenant config and return 404/403 for those endpoints for that tenant.

5. Developer docs for Dropbox plugin: We'd include in `docs/` a section such as `docs/integrations/dropbox.md` describing how the integration works, how to set up Dropbox API keys in config, etc. This acts as a playbook for operators enabling the feature and a reference for developers who might create similar integrations. It would list the environment variables needed (e.g., `DROPBOX_APP_KEY`, `DROPBOX_APP_SECRET` to be provided in `config.py` or via secrets). It would also describe any steps to link accounts (e.g., "Users go to Settings > Integrations to link their Dropbox account, which triggers OAuth..." if we had a UI for that – likely we would have added a button in `DropboxPage` to initiate linking).

Through this example, we see how a plugin is added with minimal changes to the core (just adding it to an enabled list). The plugin encapsulates its UI, API, and any background tasks. It interacts with core services (auth, config, DB) but remains mostly self-contained. We can similarly add other plugins (e.g., Google Drive integration, or a CRM integration, etc.) following this pattern. The system's design ensures we can **add, remove, or update plugins with little impact on the rest of the system** [12], fulfilling the promise of a rapidly reconfigurable architecture.

Developer Experience and Automation

Building a platform with many moving parts can be complex, so we place strong emphasis on **developer experience (DX)** to make development and maintenance efficient and error-free. Several measures in our scaffold support this:

- **Consistent Scaffolding & Templates:** We provide a cookie-cutter or CLI tool (similar to Backstage's `yarn new` command [20]) to generate boilerplate for a new plugin. For example, running a command to create a plugin would scaffold the folder in `app/plugins/` with an empty router, stub functions, and in `packages/` a stub React component and manifest entry.

This ensures every plugin starts with the correct file structure and interface, reducing guesswork and variance between plugins.

- **Documentation & Playbooks:** The `docs/` directory contains guides for common tasks. For instance, "How to create a new UI component," "How to add a new integration plugin," "Deployment playbook," etc. A new developer can follow step-by-step recipes to accomplish tasks. We maintain a **component index** (automatically via Storybook) and also a **plugin index** – a document listing all existing capability plugins, their purpose, owners, and key integration points. This is helpful for understanding the system at a glance. Furthermore, a **naming guide** is included to enforce consistency (e.g., recommending plugin modules be named `<feature>_plugin`, database tables prefixed by feature, etc.).

- **Automated Testing & Linting:** The monorepo is set up with a single test runner (for Python, using pytest; for JS, using Jest or Vitest) that can run all tests across packages. We encourage writing tests for each plugin in isolation, and integration tests for the overall system (e.g. spin up the FastAPI app with some plugins enabled and simulate typical workflows). Linting and formatting are enforced via pre-commit hooks for both Python (using tools like Ruff/Flake8, Black) and JS/TS (ESLint, Prettier). This keeps code quality high and consistent. **CI pipelines** will run these checks automatically on PRs. Because everything is in one repo, it's easier to run one CI workflow that covers frontend, backend, and shared code together, ensuring nothing is broken by a change.

- **Storybook & MCP for AI agents:** As mentioned, Storybook is not only a design/development tool but also aids in making the project **"agent-friendly."** The Storybook MCP addon (Model Context Protocol) can expose component metadata [4] which means a documentation agent or AI could query available components and even generate code using them. This aligns with our goal of developer agents – for example, an AI co-pilot could use the MCP server to list all components and their props to assist a developer in using them correctly. We treat our Storybook as the living source of truth for UI, and possibly extend it with notes on usage patterns (like a recipe book for common UI flows). Similarly, we ensure our API documentation is always up-to-date (FastAPI's interactive docs) so that whether it's a human or an AI consuming it, they get accurate information. FastAPI automatically generates OpenAPI docs for all routes, including plugins. We merge plugin docs into the main schema (since we use `include_router`, FastAPI does this by default; if we used sub-apps, we have a script to merge specs as referenced in other projects [21]).

- **Playbooks for Ops:** Multi-tenant systems and feature-flagged deployments can be tricky to manage. We include operational playbooks (likely in `docs/playbooks/`) for tasks like: adding a new tenant, enabling/disabling a feature for a tenant, performing a database migration for a plugin, rolling back a plugin, etc. These playbooks mean that not only developers but also DevOps engineers have clear guidance. For example, if a new plugin requires new environment variables (API keys), the deployment checklist (in the plugin's README or a centralized config doc) will highlight that.

- **Isolation and Independence:** Although everything is in one repo, we maintain **logical boundaries**. This modular design means a developer working on the "analytics" plugin, for example, mostly stays within `analytics_plugin` code. The interactions with core or other plugins are via well-defined interfaces (auth, common services). This prevents the "big ball of mud" problem often associated with monoliths [22] [23] – here, the monolith is *modular* by design. It is even feasible (though not necessarily planned) that a plugin could be later extracted

into a microservice if needed, because it has a clear API boundary. We note this as a strategic flexibility: **build modules now, have the option to scale out later** [24] if certain components need independent scaling. Our architecture thus "future-proofs" the system to some extent, without the upfront complexity of microservices.

- **Performance considerations:** Using one process avoids network overhead between services, but we still must ensure one plugin doesn't bottleneck others. We use Python's async features (FastAPI is async) for concurrency where appropriate (e.g., calling external APIs like Dropbox can be `async` to not block other requests). We could also use worker processes for heavy background tasks (Celery). The monorepo approach allows sharing a single pool of workers for all tasks, but logically partitioning tasks by plugin so we know which feature might be consuming resources. Monitoring can be tagged by plugin (e.g., include plugin name in metrics like request count, error count).

In summary, this scaffold is designed to be **comprehensive and developer-friendly**. New features can be added as plugins by following a clear specification, leveraging shared resources (UI components, SDK, auth) provided by the core, and without entangling with other features. The monorepo ensures a single source of truth and easy code reuse, at the cost of repository size which is mitigated by using workspace tools and a structured approach. By adhering to plugin architecture principles – *extensibility, modularity, and separation of concerns* – we enable the platform to evolve quickly and safely [18] [19].

This reference architecture can serve as a starting point for building a robust SaaS platform that needs to rapidly enable or disable features, customize per tenant, and grow with a developer team (and their AI agents) that can navigate and contribute to the code with confidence. Each part of the stack (React frontend, FastAPI backend) complements the other, connected by shared contracts and schemas (e.g., TypeScript types in SDK generated from FastAPI's OpenAPI, so front-end and back-end stay in sync). Ultimately, the architecture emphasizes **plug-and-play capabilities** – much like LEGO blocks as one author described [11] – allowing the system to be reconfigured on demand to meet changing requirements.

Sources:

- Monorepo code sharing benefits [2]
- Backstage plugin structure and naming conventions [25] [5]
- Plugin architecture design pattern and core/plugin separation [1] [10]
- Plugin registration via configuration (manifest) [26] [7]
- Storybook MCP for AI agent integration [4]

---

[1] [7] [10] [12] [15] [17] [26] Plug-in Architecture. and the story of the data pipeline... | by Omar Elgabry | OmarElgabry's Blog | Medium
https://medium.com/omarelgabrys-blog/plug-in-architecture-dec207291800

[2] [3] A Guide to Monorepos for Front-end Code | Toptal®
https://www.toptal.com/front-end/guide-to-monorepos

[4] Addon MCP | Storybook integrations
https://storybook.js.org/addons/@storybook/addon-mcp

[5] [6] [20] [25] Building Frontend Plugins | Backstage Software Catalog and Developer Platform
https://backstage.io/docs/frontend-system/building-plugins/index/

8 9 13 16 21 22 23 24 GitHub - YoraiLevi/modular-monolith-fastapi: This is a modular monolith Fast API project that uses the latest and greatest tooling (uv, ruff, pyright, pydantic, pytest, fastapi, sqlmodel, etc) attempting to implement a modular monolith architecture. The repository include pre-commit hooks for ruff, pyright, and uv.
https://github.com/YoraiLevi/modular-monolith-fastapi

11 How I Built a Plugin-Driven FastAPI Backend That Auto-Registers Routes | by Bhagya Rana | Medium
https://medium.com/@bhagyarana80/how-i-built-a-plugin-driven-fastapi-backend-that-auto-registers-routes-e815a7298c29

14 18 19 Plugin Architecture Design Pattern – A Beginner’s Guide To Modularity
https://devleader.substack.com/p/plugin-architecture-design-pattern?utm_campaign=post&utm_medium=web