Plugin-based Application Architecture

Below is an architecture diagram (Mermaid) and supporting notes showing how a core app and plugins interact.

```
flowchart TB
  subgraph Backend
    Core[Core Services]
    Core --> Auth[Auth Service\n(Login / JWT / OAuth)]
    Core --> Perms[Permission Engine\n(Role / Object / Action)]
    Core --> Menu[Menu Registry]
    Core --> PL[Plugin Loader]
    Core --> API[API Gateway / Router]
    Core --> DB[(Database)]
    PL --> PR[Plugin Registry & Metadata]
    PL --> Migrate[Migration Runner]
    API --> DB
    Auth --> DB
    Perms --> DB
  end

  subgraph Plugins
    P1[Plugin: Contacts]
    P2[Plugin: Invoicing]
    P3[Plugin: Reporting]
    P1 --> P1_mod[models.py / migrations]
    P1 --> P1_api[routes / controllers]
    P1 --> P1_ui[ui bundle / components]
    P1 --> P1_meta[plugin.json, permissions.json, menu.json]
    P2 --> P2_meta[...]
    P3 --> P3_meta[...]
  end

  subgraph Frontend
    Shell[Frontend Shell (SPA)]
    Shell --> AuthUI[Auth UI]
    Shell --> MenuUI[Dynamic Menu]
    Shell --> MFLoader[Micro-frontend Loader / Plugin Loader]
    MFLoader --> P1_ui
    MFLoader --> P2_ui
  end

  Core ---|loads| PL
  PL ---|reads| PR
  PL ---|runs| Migrate
```

```
PR ---|registers| Menu
PR ---|registers| Perms
PR ---|mounts| API
API ---|exposes| P1_api
Shell ---|requests| API
Shell ---|fetches| Menu
DB ---|stores| PluginData[Plugin Data Schemas]
PluginData ---|owned-by| P1_mod

classDef core fill:#f9f,stroke:#333,stroke-width:1px;
class Core,Auth,Perms,Menu,PL,API,DB core;
```

Components & Responsibilities

- **Core Services**: Authentication, permission enforcement, menu registry, API gateway, and plugin loader.
- **Plugin Loader**: Discovers plugin packages (file system or package registry), validates manifests, registers permissions & menu entries, runs migrations, and mounts routes/UI.
- **Plugin Package (example: `contacts`)**:
- `models/` + migrations: DB schema owned by plugin
- `api/`: backend endpoints
- `ui/`: frontend bundle or micro-frontend
- `permissions.json`: permission definitions (e.g. `contacts.view`, `contacts.create`)
- `menu.json`: menu entries to add to the core menu
- `plugin.json`: metadata (name, version, entrypoints)
- **Frontend Shell**: minimal SPA shell that authenticates users, renders menus based on permissions, and lazy-loads plugin UIs (via module federation, iframe, or dynamic import).
- **Database**: Can be a single DB with namespaced tables per plugin, or multiple DBs if isolation is required.
- **Migration Runner**: Executes plugin migrations in a safe order (core first, then plugins by dependency).
- **Event Bus (optional)**: For cross-plugin communication (publish/subscribe) and async jobs.

Startup / Registration Flow

1. Core boots and initializes DB, Auth, Perms, Menu.
2. Plugin Loader scans configured plugin locations.
3. For each plugin: validate `plugin.json`, register metadata in Plugin Registry, register declared permissions and menu entries, run migrations, and mount API routes.
4. Frontend Shell fetches menu entries and permission info for the logged-in user and displays the appropriate menu.
5. When user navigates to a plugin route, the Shell lazy-loads the plugin UI bundle and communicates with plugin backend endpoints.

Notes & Trade-offs

- **Security**: Core enforces all permission checks; plugins should call core permission APIs or use middleware.
- **Versioning**: Plugins must declare compatibility (core version) in `plugin.json`.
- **Isolation**: For stronger isolation, run plugin services in separate containers communicating over API/gRPC.
- **Frontend strategy**: Module Federation (Webpack 5), iframe-based isolation, or a single-build with dynamic imports.

If you want, I can also: - provide a simplified sequence diagram for plugin load lifecycle, - produce a concrete file/directory scaffold for the core + contacts plugin, - or convert this diagram into a PNG/SVG you can download.