ChatGPT

EdgeX UI Migration Guide

The EdgeX Foundry platform is built as a **layered microservice architecture**. At the core are **Core Services** (Core Data, Core Metadata, Core Command, plus configuration/registry), surrounded by **Support Services** (Logging, Notifications, Scheduling, Rules Engine) and **Application/Export Services** (application-function pipelines, distribution, client registration). A **System Management** layer (System Management Agent and management API) and **Security** layer (API Gateway, Secret Store) provide operational control and secure access. *Figure 1* illustrates the overall EdgeX/Xpert architecture. Each UI feature corresponds to one or more of these backend services [1] [2].

Figure 1: EdgeX microservice architecture by layer (Device Services, Core, Supporting, Application/Export, Security/System) [1] [3].

1. Essential Backend Microservices (by layer)

To support **all UI functionality**, the custom backend must replicate the following EdgeX microservices:

- **Core Services:**
- **Core Metadata:** Manages *devices, device services, device profiles, provision watchers, and custom properties*. All "Metadata" UI pages (Device Services, Devices, Profiles, Watchers) use Core Metadata APIs to list, add, update or delete objects [2] [4].
- **Core Data:** Stores *Events and Readings* from devices. The Dashboard and Data Explorer use Core Data to query recent sensor data [5]. If Core Data is optional (EdgeX 2.x+), the backend must at least expose current event data (or aggregate metrics) for charts.
- **Core Command:** For *executing device commands*. The UI's Device page lets users "try" GET/SET commands on devices (via Core Command) [6]. Core Command is the entry point for all device control requests from the north side.

- **Configuration/Registry:** EdgeX uses a central registry (Consul) and *Config Seed* service to distribute configuration profiles. While not a user-facing service, the backend needs **config/registry support** (e.g. Consul or equivalent) to manage and expose service configurations if required by the UI (the UI "Config" screens display JSON from the registry) [7].

- **Device Services Layer:**

- **Device Service Registry (metadata):** The UI shows a list of *registered Device Services* (from Core Metadata) and allows changing their admin state [4]. The backend must handle listing device services and updating their info via Core Metadata APIs.

- *Physical Device Drivers:* Typically EdgeX has Go-based device service apps (e.g. Virtual, Modbus, OPC-UA). The UI itself does not "implement" these; it only displays existing device services and their status. For completeness (e.g. adding a Virtual Device to test), the backend project can include or

reuse EdgeX's device service SDKs or example services, but the UI features only require the registry metadata and command interfaces of device services.

- **Supporting Services:**

- **Support Scheduler:** Manages *Intervals/Schedules*. The UI's "Intervals" page (or "Scheduling") lists and edits schedule triggers; this uses the Scheduler service APIs.
- **Support Notifications & Alerts:** Manages *Notification and Alarm rules*. The UI's "Notifications" page lists recent notifications (from Support Notifications). The UI can also create/clear alarms (often via core metadata+notifications).
- **Support Rules Engine:** Manages *rules and pipelines*. The UI's "Rules" page allows viewing/editing rule-engine pipelines (using App Service Configurable or eKuiper under the hood). The backend should provide rule CRUD and status, typically via the Rules Engine service or the App Functions SDK.

- **Support Logging:** While the UI has a "Logs" view, this often reads from centralized logging (e.g. files or a log service). EdgeX has a **Support Logging** service (and edges like Logstash). At minimum, the backend may need to aggregate service logs (or proxy to wherever logs are kept) if the React UI includes a log viewer.

- **Application/Export Services:**

- **Application Services:** In EdgeX this means functions pipelines (Configurable App Service). The UI can display and manage the *pipelines and exporters* configured in the App Service. The backend should include at least the app-service-configurable logic and its REST endpoints (or implement equivalent endpoints) so the UI can list and update function pipelines, exporters, and triggers.

- **Export/Distribution Services:** EdgeX has services like Distribution (for binary data, IoT Core) and Client Registration. The official Angular UI does not currently expose these, but they are part of the export layer. If any pipeline configuration or status is shown in UI, ensure the corresponding export-service endpoints exist.

- **System & Security Services:**

- **System Management Agent (SMA):** Manages service lifecycle, config and metrics. The UI's "Services" page (start/stop/restart buttons, service metrics) calls the SMA's **management API** [8] [9]. The backend must include a System Management component (like edgexfoundry/edgex-go's SMA) to respond to these control requests.
- **Registry & Service Discovery:** Consul or a similar registry is used so UI can discover services and their health. The new backend should also register its microservices and allow queries of service status (through SMA or a custom API).
- **Security (API Gateway / Secret Store):** If running secure EdgeX mode, the UI uses an API Gateway (NGINX) and Vault token for auth [10]. The custom backend should integrate with EdgeX's security (or its own equivalent auth system), but UI compatibility mainly needs *token-based auth on each route*. In practice, ensure all new API routes enforce the same JWT or API-key checks as EdgeX services do under the gateway.

Key microservice summary (by category):
- *Core:* **core-data**, **core-metadata**, **core-command**, (plus **config-seed/registry**).

- *Device:* **(various device services)** – these are not reimplemented but must be listed/queried via Core Metadata. Optionally include an example (virtual) device.
- *Support:* **support-logging**, **support-notifications**, **support-scheduler**, **support-rules-engine**.
- *Application:* **app-service-configurable** (with functions SDK), **distribution** (optional).
- *System/Security:* **system-management-agent**, **service-management** API, **registry-consul**, **secret-store-init** (Vault), **API gateway**.

Each of these corresponds to an EdgeX repository or component. For example, Core Data/Metadata/Command are implemented in edgexfoundry's Go services (edgex-go monorepo), while support and app services have their own repos (e.g. support-scheduler, app-service-configurable, etc.). All of these existing EdgeX services can be used as references or included as modules in the new backend project.

2. Repositories and Shared Go Modules

The custom backend should leverage EdgeX's existing code and Go modules wherever possible. Key code bases and libraries include:

- **edgexfoundry/edgex-go (Core/Support Monorepo):** Contains the Go implementations of core services (core-data, core-metadata, core-command) and many support services. Fork or vendor this repo (or relevant parts) for core functionality.
- **Application Services Repos:** Include the Configurable App Service code for pipeline management (edgexfoundry/app-service-configurable-go) and any distribution services if needed.
- **Device Services (optional):** For example, edgexfoundry/device-virtual-go (a sample virtual device driver) can be included if device simulation is desired, though not strictly required for UI.
- **Go Modules (shared libs):** The following EdgeX Go modules should be imported/used in the new project (per edgex-go Release notes[62†L395-L402]):

- `github.com/edgexfoundry/go-mod-core-contracts` – **Data models and REST contract definitions** for all EdgeX objects (devices, events, readings, etc). Use this for DTOs and API clients.
- `github.com/edgexfoundry/go-mod-bootstrap` – **Bootstrap framework** for EdgeX services (common main(), config loading, startup checks, registry integration). Using bootstrap ensures consistent initialization (as done in EdgeX) [11].
- `github.com/edgexfoundry/go-mod-messaging` – Abstraction for **message bus** (Redis/MQTT) if needed for event publishing.
- `github.com/edgexfoundry/go-mod-registry` – **Configuration/Registry client** (for Consul or other registry).
- `github.com/edgexfoundry/go-mod-secrets` – **Vault/Secret-store integration**, if using EdgeX Vault for security.
- `github.com/edgexfoundry/go-mod-configuration` – Configuration loading (listed as indirect dependency of edgex-go) [12].
- `github.com/edgexfoundry/go-mod-logging` – If needed, though bootstrap provides logging clients via go-mod-core-contracts too.

These modules provide the plumbing so you can focus on Echo handlers and business logic. For example, the go-mod-bootstrap framework will connect to Consul, Vault and set up the common middleware

(logging, metrics) automatically. EdgeX v3.0 has moved **all services to use the Echo router** and consolidated common code into go-mod-bootstrap [11], so it is recommended to follow that pattern.

In practice, your `go.mod` should include at least the above modules (and versions matching your EdgeX release). Many are referenced by edgex-go itself, as shown in the edgex-go release notes [13]. For any EdgeX API client (e.g. querying Core Metadata), you can use the **go-mod-core-contracts REST clients** or write your own Echo handlers using the shared contracts package.

3. UI Interactions → Backend APIs

The existing Angular UI calls REST APIs on EdgeX to implement each feature. When migrating to React with a custom Go/Echo backend, you should **mirror those interactions** so the React UI can function with minimal change. Key mappings include:

- **Device Services & Devices:** The UI lists device services and devices via Core Metadata APIs. E.g. GET `/core-metadata/api/v2/device-service` and GET `/core-metadata/api/v2/device`. In the new backend, implement equivalent routes (e.g. `GET /api/v2/device-service` and `/api/v2/device`) that internally call Core Metadata logic or database. Creating a device via the UI sends a POST to `/device` (device profile ID, protocol properties, etc); your backend should handle this and persist via Core Metadata. *Mapping:* UI → Echo route → CoreMetadataClient (or DB call) → return JSON to UI.

- **Device Profiles:** Similar to devices. The UI allows adding/editing profiles (POST/PUT `/deviceprofile`). The backend needs endpoints for listing (`GET /deviceprofile`), creating, updating and deleting profiles, using the Core Metadata service under the hood.

- **Device Commands:** On the Device page, "Try" actions invoke Core Command. The UI calls e.g. `GET /core-command/api/v2/device/{name}/command/{cmdname}` or `PUT /core-command/api/v2/device/{name}/command/{cmdname}`. In your backend you should expose `/api/v2/device/{id}/command/{cmd}` that performs the appropriate GET/PUT against the device. Internally this means constructing a Core Command request (e.g. using go-mod-core-contracts models) and dispatching to the device service.

- **Schedules (Intervals):** The UI "Intervals" page uses Support Scheduler APIs. For example, `GET /support-scheduler/api/v2/intervals` to list schedules. Replicate these in the backend (`/api/v2/intervals`) that read/write the scheduler database.

- **Notifications/Alerts:** The UI "Notifications" page calls Support Notifications APIs (`GET /support-notifications/api/v2/notification`). Implement routes like `/api/v2/notifications` in your backend mapped to the same store.

- **Rules Engine:** The UI "Rules" page manages pipeline rules (e.g. sequence of filters/exports). This uses the Application Services REST (Configurable App Service) or eKuiper. Provide API endpoints (e.g. `/api/v2/rules` and `/api/v2/filters`) that let the UI CRUD these objects. The new backend can reuse the App Functions SDK or similar to process rule definitions.

- **Application Services:** If the UI shows App Services (e.g. Configurable App pipeline), implement endpoints to list, create, update the pipeline and its subscription. These correspond to the existing App Service REST (see docs). Use the app-service-configurable-go logic or its contracts.

- **System Management (Services & Config):** The UI "Services" list calls the System Management Agent: e.g. `GET /api/v2/registry` or `/api/v2/health`. The SMA also accepts PUT (control) operations. Your backend should include an SMA with Echo routes for:

- `/api/v2/health` or `/api/v2/registry` to report status of all services (including your new ones).
- `/api/v2/command/service/{name}/stop|start` to stop or start a service on the host (via exec). These routes ultimately call EdgeX's management API (or your own).

- `/api/v2/metrics/{name}` to return memory/CPU metrics (collected by the SMA).
  EdgeX's System Management API can guide these endpoints.

- **Data/Events:** For the Dashboard charts, the UI will need recent event data from Core Data (`GET /core-data/api/v2/event`). Provide routes in the backend that query the events database and return JSON. If Core Data is optional, you may use the message bus or a lightweight store to accumulate recent readings for display.

In summary, **every UI AJAX call must be backed by an Echo handler** exposing the corresponding RESTful URL. You should generally preserve the same URI patterns (and JSON schemas) as EdgeX Core/Support services so the UI code needs minimal change. Internally, these handlers can either delegate to the imported EdgeX service code (calling its client methods) or reimplement the logic against a database. For example, you could fork `core-metadata-go` and run it within your Echo router. The key is to satisfy the UI's contract (request/response shape) for each feature.

4. Designing the Go/Echo Backend

Service Structure: Follow EdgeX's microservice separation. In a monorepo or separate repos, create one Go service per logical component (core-metadata, core-data, core-command, etc.). Use Go modules to manage dependencies for each. A common pattern is to have:

```
/cmd/{service-name}/main.go    – service entrypoint using go-mod-bootstrap
/internal/{service-name}/     – business logic (handlers, persistence)
/pkg/{common}                 – shared code (models, middleware)
/go.mod                       – root module or per-service modules
```

Each `main.go` initializes an Echo HTTP server via `bootstrap.Run()` [11]. Configure it to load service-specific settings (from TOML/Consul) and to register API routes. Because EdgeX v3 replaced Gorilla Mux with **Echo** for all services [11], you should define all REST routes with Echo (router groups, middleware, etc.). For example, in `cmd/core-metadata/main.go`, bootstrap will load the config and then you do `router.GET("/api/v2/device", metadataHandler.ListDevices)` etc.

Shared Modules: Keep shared code (e.g. logging setup, common response formats, error handling) in a module like `go-mod-bootstrap`. In fact, EdgeX's new code has moved common APIs (e.g. `common.ApiVersionHandler`, `common.PatchHandler`, etc.) into bootstrap [14]. Reuse these by importing go-mod-bootstrap. Also use `go-mod-core-contracts` for all DTO types and for its client libraries (e.g. `clients/coredata`).

Configuration: Use a combination of local config and registry (Consul). Store service configs (port, database URLs, message bus, etc.) in a central config store. `go-mod-registry` can read from Consul. Typically each service's config lives in `/res` TOML files (as in edgex-go) or in Consul KV. Ensure the React UI's backend has its own config (e.g. listening port).

Persistence: Core Metadata/Data/Notifications/Scheduler all normally use databases (Mongo or Redis). You can reuse EdgeX's persistence layers by running their code (in-process or as separate Docker) or by migrating data into your own database. For a smooth migration, it's easiest to run the existing EdgeX database (MongoDB/Redis) and have your services use the same schema (via the EdgeX data models) so no data is lost.

Shared Clients: For cross-service calls, you can use the generated client libraries in go-mod-core-contracts (e.g. `restclient.NewCoreMetadataClient(...)`) to call between your Echo services (if needed). However, if you deploy them on the same host, you might skip HTTP and call the internal logic directly.

Example Project Layout (suggested):

```
edgex-backend/
├── cmd/
│   ├── core-metadata/
│   ├── core-data/
│   ├── core-command/
│   ├── support-notifications/
│   ├── support-scheduler/
│   ├── support-rules-engine/
│   ├── app-service-configurable/
│   ├── system-agent/
│   └── secrets-setup/
├── internal/     (per-service business logic packages)
├── pkg/          (shared utilities, if any)
└── go.mod        (with required EdgeX modules e.g. go-mod-core-contracts, go-
mod-bootstrap, etc.)
```

Each service's `main.go` uses something like:

```
func main() {
    service := bootstrap.NewService(
        bootstrap.WithServiceKey("core-metadata"),
        bootstrap.WithConfigProvider(...),
```

```
        // other options...
    )
    service.Run(run)  // where run sets up Echo and routes
}
```

This pattern mirrors EdgeX's own services. See the EdgeX Configuration[9†L55-L63] docs for examples of service folders.

5. Gradual Migration and Validation

To migrate safely, rebuild services **one layer at a time** and continuously test the React UI against the new backend:

1. **Start with core metadata:** Implement or deploy your Echo-based Core Metadata service first. Point the React UI (temporarily) to your new endpoints (e.g. via a proxy or config change) and verify all Device/Device Profile/Device Service UI pages work (add/edit/delete devices, list profiles). Keep Core Data, Command, etc. still pointing at the old services.

2. **Add Core Data & Command:** Next, bring up your Core Data and Core Command equivalents. Verify the Dashboard and Data Explorer charts (events/readings) come from your Core Data, and that device commands from the UI now hit your Core Command layer (test the "Try GET/SET" on devices).

3. **Add Support Services:** Implement the Scheduler, Notifications, and Rules Engine endpoints. Confirm the Intervals, Notifications, and Rules pages in the React UI function as before. If something breaks, you can toggle between old/new services for troubleshooting.

4. **Integrate SMA/Management:** Deploy the System Management Agent with your new Echo-based endpoints for service control and health. Use the UI to start/stop a service and check metrics to validate this part.

5. **Test Security Flow:** If running secure mode, ensure your new API routes enforce the same Vault/Nginx token flow. Use the UI's login/token prompt to test access to your new endpoints.

Throughout, maintain compatibility by keeping the **request and response schemas identical** to what the UI expects. Utilize the EdgeX API docs or Swagger/OpenAPI definitions to verify input/output formats. Automated testing can help: consider writing integration tests (or using EdgeX's existing black-box test suite) against your new services as each is rolled out.

Finally, run the EdgeX **compose deployments** (or your Kubernetes setup) with a mix of old and new services to validate interoperability. The gradual approach ensures you can fall back (keep old service for any UI feature) until the replacement service is fully proven.

References: EdgeX's architecture and service breakdown [1] [3] informed the service list. The EdgeX GUI documentation confirms which objects the UI manages [2]. The EdgeX Go modules and bootstrap usage

are documented in release notes [13] [11]. (See also the IOTech Edge Xpert diagram in *Figure 1*.) These sources guided the service grouping, module selection, and design recommendations above.

---

[1] [3] 3. EdgeX Foundry Microservices Architecture — EdgeX documentation
https://fuji-docs.edgexfoundry.org/Ch-Architecture.html

[2] [4] [5] [6] [7] [8] [9] Graphical User Interface (GUI) - EdgeX Foundry Documentation
https://docs.edgexfoundry.org/4.1/getting-started/tools/Ch-GUI/

[10] API Gateway - EdgeX Foundry Documentation
https://docs.edgexfoundry.org/3.2/security/Ch-APIGateway/

[11] [12] [13] [14] Releases · edgexfoundry/edgex-go · GitHub
https://github.com/edgexfoundry/edgex-go/releases