ChatGPT

Designing a Common Modular Backend for Location Tracking Apps

Introduction

Building multiple applications that **track the location of various "objects"** can be vastly simplified by creating a **common, modular backend platform**. Instead of reinventing the wheel for each use case – whether it's wildlife photos from trail cameras, attendees at a conference, neighborhood incidents, or family travel logs – we establish a single shared backend. This core platform will handle the fundamental tasks of storing objects and their locations over time, while allowing each app to have a custom frontend experience. The goal is to **"build once, reuse many times,"** much like a white-label SaaS approach where identical core functionality is leveraged across domains [1]. By sharing one backend, app developers can focus on their domain-specific features and UI, confident that the heavy lifting of data management, geospatial tracking, and scalability is already solved centrally.

Unified Core Data Model

At the heart of this platform is a **unified data model** that represents any trackable object and its location history. This common schema acts as a "shared language" for all applications, ensuring consistency across different use cases [2]. In practice, we define generalized entities such as:

- **Tracked Object** – the item or thing being tracked (e.g. a specific trailcam photo, a person or badge ID, an incident report, a visited place). Each object would have a unique ID, a human-readable name/description, and metadata common to all objects (perhaps a creation timestamp, etc.).
- **Location Record (Event)** – an entry capturing the object's location (geographical coordinates) and a timestamp. Each record links to a Tracked Object, forming a time-series of where/when that object was observed. For example, a trailcam photo object might have several Location Records if the camera moved locations or if multiple sightings are grouped; a conference attendee object could have Location Records for each check-in location and time.
- **Tenant/Application** – a context to segregate data by app owner or client. This could be an implicit field (e.g. an "AppID" or tenant ID stored with each object) to ensure data isolation between, say, a family's travel log and a bank's inventory tracker. Every data row is tagged with which tenant it belongs to [3]. This allows a single database/schema to serve multiple apps securely while still partitioning their data logically.

By **standardizing core entities and relationships** like this, we make it possible for multiple applications to share and understand data easily [4]. Each application might introduce some custom attributes (for instance, an "incident severity" field or an "animal species" field), but those can be layered on top of the core fields. The common data model provides the foundation (ID, location, time, etc.), and it is **extensible** to accommodate app-specific details. This approach is analogous to Microsoft's Common Data Model, which offers a set of standardized, extensible schemas (entities, attributes, relationships) to represent common

concepts while allowing extension for specific needs [5] [6]. In our case, the concept of a "location-tracked object" is the common denominator across all apps.

Modular Backend Architecture

To implement this in a robust way, the backend should be designed in a **modular, service-oriented fashion**. Rather than one giant monolithic server doing everything, we can break the backend into logical components or microservices. This modularity makes the system easier to maintain and lets us scale or modify parts of the system independently. Key modules might include:

- **Object Management Service:** Handles creation, updating, and retrieval of tracked objects (the "what" we're tracking). This service knows about the core object schema and enforces the common fields (and perhaps validates custom fields).
- **Location Tracking Service:** Focuses on recording location events and querying them. It would provide APIs to add a new location record for an object, and to query location histories or find objects near a given location. This service would encapsulate any geo-specific logic (e.g. coordinate transformations, spatial indexing).
- **User & Auth Service:** Manages user accounts, authentication, and authorization. If app owners or end-users need to log in, this service ensures they can only access their tenant's data. A single sign-on system could allow one user identity to access multiple of the apps if needed, but with permissions scoped per app. (This follows the principle of centralizing authentication but segregating authorization for each app's data [7] [8].)
- **Media or Attachment Service:** If the objects include photos or other files (like trailcam images or incident photos), a separate module can handle file storage (e.g. interfacing with cloud storage for saving images, and storing URLs or IDs in the database). This keeps heavy binary data management decoupled from the core database logic.
- **API Gateway / GraphQL Layer:** A unified API entry point that client apps communicate with. This could aggregate the various services behind the scenes and present a clean RESTful API or GraphQL interface to developers building the frontends. The API layer can also implement a **Backend-for-Frontend (BFF)** pattern if needed – for example, if each app needs a slightly different API design, the gateway can shape the responses accordingly [9] [10]. However, if the data model is truly common, a one-size-fits-all API may suffice with clients simply ignoring irrelevant parts.

Each module is developed as a **reusable component**. For instance, if we improve the Location service (say, adding a new way to filter data by region or a new mapping feature), all apps benefit immediately. Likewise, bug fixes or performance improvements in the core will propagate to every application using it. This architecture embodies the **"single codebase, multi-app"** philosophy – much like how a shared library of UI components can be reused across apps for consistency [11] [12], here we have shared backend components for all tracking apps. It's important to version and manage these modules carefully; updates should be backward-compatible or selectively rolled out so as not to break one app while updating another. With clear module boundaries and well-defined interfaces (APIs), each app can plug into the common backend with minimal coupling.

Multi-Tenancy and Data Isolation

Because we expect **multiple independent applications (tenants)** to coexist on this backend, multi-tenancy is a core concern. The simplest approach is a **shared database & schema** for all tenants, distinguished by a

tenant ID on each record [3]. For example, the "objects" table and "locations" table would each have a column for `TenantID`, and every query in our services will filter by the tenant so that apps never see each other's data. This model is cost-efficient and allows maximal reuse of infrastructure (one set of servers, one DB for all) [13]. However, it requires careful implementation of security in the application layer – we must rigorously enforce tenant scoping in queries and APIs so no data leaks across boundaries, even in edge cases or under attacks [14].

For most scenarios (like a family app or a typical business), this **shared-schema multi-tenancy** is appropriate, since it can serve many tenants on a small number of servers [15]. It's the model used by many large SaaS platforms to serve thousands of customers efficiently. The trade-off is that tenants share resources, so extremely heavy usage by one tenant could affect others if not managed (we can mitigate this with rate limiting or by scaling out the infrastructure). Also, some clients (e.g. in banking or healthcare) might demand stricter data isolation for compliance [16]. In such special cases, the architecture can support **alternative tenancy models**: for example, deploying a separate database schema (or even a separate database instance) per tenant [17] [18]. A dedicated schema per tenant offers a middle ground – data is siloed per client but the application code is still shared. A fully separate deployment per tenant (single-tenant model) is also possible but only worth the cost for a handful of large customers due to high overhead [19] [20].

Importantly, our design keeps the **tenant concept as a first-class element**. The backend's services always take the tenant context into account, and the system could even allow configuration per tenant (e.g. custom branding or settings for each app) [21]. Tenant-aware design considerations include:
- **Security:** Use the tenant ID in authorization checks everywhere. Multi-tenant SaaS must ensure one tenant cannot access or infer another tenant's records [14]. Often, an additional filter or schema qualifier is added to every DB query. Some databases support row-level security policies which we could leverage to automatically enforce `TenantID = currentTenant` on queries.
- **Scaling:** We should monitor if certain tenants grow very large. Techniques like **partitioning data by tenant** or assigning heavy tenants to their own DB nodes can be employed if needed (for example, by "sharding" on tenant ID for horizontal scaling). The system might maintain a registry mapping each tenant to the infrastructure that holds their data [22] – this way, we can dynamically distribute load as new tenants onboard or existing ones expand.
- **Lifecycle and Isolation:** Backup/restore procedures should accommodate per-tenant operations. In a shared DB, restoring a single tenant's data is tricky (involves extracting just that tenant's rows) [23]. If a tenant leaves, we need to delete or archive their data safely without impacting others. Planning for such operations ensures maintainability in the long run.

Overall, multi-tenancy enables **serving many clients from one codebase** cost-effectively, but it demands that we architect for **tenant isolation, security, and scalability** from the start. When done right, tenants get a seamless experience as if they each had their own application, while the provider (us) only maintains one platform.

Extensibility for Different Object Types

One of the biggest challenges in a one-size-fits-all backend is **handling the custom data needs** of each application. Our core model knows about "objects" and "locations," but each use case will want to store different attributes about those objects. For instance, a *trailcam photo* might have an animal species and a link to an image; a *conference attendee* has a name, affiliation, maybe a badge number; an *incident report*

has a description, category, and status; a *visited place* might have a rating or travel notes. We cannot simply predefine a giant object schema with dozens of optional columns for every possible attribute – that would be sparse, hard to maintain, and still inevitably miss some future requirement. **Hardcoding all variations is infeasible [24].** Instead, the backend must be **extensible**, allowing each tenant or app to define custom fields or even custom object sub-types on top of the common core.

There are a few design patterns to achieve this flexibility:

- **Entity-Attribute-Value (EAV) or Dynamic Fields Table:** A traditional relational solution is to have a separate table to store custom fields as key-value pairs linked to the object. For example, one might have tables like `ObjectBase` (with core columns) and `ObjectAttributes` (with columns: ObjectID, FieldName, FieldValue) [25]. A related `FieldDefinition` table can define the schema of those fields (data type, label, etc.) per tenant [26]. This approach keeps the core schema stable and puts variable data in rows rather than columns. It maintains referential integrity and works with SQL constraints [27]. However, it comes with **downsides**: queries become complex and potentially slow (lots of JOINs or aggregations to reconstruct objects) and indexing dynamic fields for fast search is harder [28]. As the number of custom fields grows, the performance can degrade and the SQL logic gets messy.

- **Pre-Allocated Flexible Columns:** Another approach some systems use is to include a handful of generic nullable columns in the main table (like `CustomText1, CustomText2, CustomNum1…`) which tenants can repurpose [29]. This avoids separate tables, but it's very limiting and not self-describing – it requires out-of-band knowledge of what each slot represents for a given tenant. If a tenant needs more custom fields than there are slots, or a different data type mix, this model breaks down [30]. It's generally seen as a clunky workaround. A variant of this is to have an **extension table** (one-to-one with the main object) where new columns can be added for a specific tenant's needs [31] [32]. That can work if tenants are few and changes are manually managed, but in an automated multi-tenant environment it's not ideal to be altering schemas on the fly for each customer.

- **Schema-less (NoSQL) Storage:** Embracing a NoSQL database (or a hybrid approach) can elegantly solve the variable schema problem. For example, using a document store like MongoDB, each object can be stored as a JSON document that includes both the standard fields and any custom fields desired. MongoDB (and similar document DBs) does not require a fixed schema – one record can have fields that another lacks – and it even supports nested structures (arrays, sub-documents) which can model complex data flexibly [33]. We could maintain a separate collection or document to describe the form (metadata) for each tenant's object type (essentially a JSON schema or field definitions) [34] [35]. The frontends can query this metadata to dynamically render forms or UI elements for the custom fields. This **metadata-driven approach** was demonstrated by Dylan Lee in a multi-tenant inventory system: the app fetched the tenant's field definitions and then generated the input form accordingly, storing the entered data as flexible JSON [36] [37]. The NoSQL route avoids the heavy JOIN cost – each object is self-contained – and is naturally suited to **heterogeneous data**. Moreover, modern document databases still allow indexing on specific fields (including geo-coordinates and even inside arrays) and can enforce some structure via JSON schema validation if needed.

- **JSON in a Relational DB:** As a compromise, one can also use a relational database (for its robust transactions, SQL features, etc.) but put the custom attributes into a JSON column. PostgreSQL, for

instance, has a JSONB column type that can be indexed and queried with JSON path expressions. This way, the base columns (like ID, tenant, core fields) are regular columns, and an `extra_attributes` JSONB column holds any tenant-specific data. This approach achieves flexibility and keeps a single source of truth (one table), but reading and filtering by those fields might not be as fast as a purpose-built document store for very large datasets. Still, for moderate use it works and allows using the power of SQL alongside.

Our platform can adopt **any of the above or even a combination**, depending on the exact needs and familiarity of the team. The **key requirement** is to *"allow customers to extend your default data model to meet their needs, without affecting other customers' data model"* [38] . In practice, the **NoSQL + metadata strategy** is very attractive for a broad, evolving product because it cleanly separates the concerns: the core backend provides generic storage and retrieval, while each tenant gets to define (via a metadata configuration) what additional fields they need. For example, tenant A (wildlife app) can add fields like `species` (text) and `imageURL`, tenant B (conference app) adds `company` and `badgeId`, etc., and these will simply exist in their objects' JSON without requiring a schema migration for others who don't need them. This method was shown to support from tenants wanting "5 fields to others wanting 50" with *no code changes per tenant* [24] – a clear win for developer productivity and scalability of the platform.

One must also implement a user-friendly way to manage these custom fields. This could be an **admin UI** where app owners define new fields (name, type, maybe choices or validation rules), which updates the metadata store. The backend uses this metadata to validate inputs and maybe to structure API responses. The frontends can retrieve the definitions to know what to display. Essentially, our backend becomes partly a **configuration-driven system**: the core is static, but it interprets dynamic schemas per tenant. This adds complexity, but it's necessary for a system intended to serve very diverse applications. The benefit is huge: adding a new type of data for one client doesn't require deploying a new service or altering database schema globally – it's an isolated change in that tenant's configuration.

In summary, **extensibility is achieved by combining a strong common core with flexible extensions**. We ensure every app has the basics (object ID, location, time, tenant linkage) for interoperability and core features, but we also empower each app to capture the information that matters for their scenario. This approach is commonly seen in platforms like CRM or ERP systems (e.g., Salesforce's platform allows custom fields per organization, stored under the hood with metadata and separate index tables [39] [40]). Adopting similar patterns will make our tracking backend capable of serving a **wide audience from a single codebase** without one app's customization interfering with another's.

Geospatial Data Handling

Since all these apps revolve around **location tracking**, robust geospatial data support is a critical part of the backend. We need to store coordinates (latitude/longitude, or even more complex geo-shapes if ever needed) and enable spatial queries (like "find all incidents within 5 km" or "list the photos near this camera"). There are two mainstream approaches here, depending on our technology stack choice, and fortunately both can be integrated with the extensibility approach above:

- **Relational + Spatial Extension:** Using a relational database that supports spatial data, such as PostgreSQL with the PostGIS extension, allows us to store location as a geometry column and query it with standard SQL. PostGIS "adds support for storing geospatial data in PostgreSQL and executing

location queries in SQL" [41]. We can define a column `location GEOGRAPHY(Point)` in the Location Record table, for example, to store a point with latitude/longitude coordinates [42]. PostGIS provides spatial indexes and dozens of functions for spatial calculations – e.g., finding distances, checking point-in-polygon, etc. An example given in documentation is creating a table and inserting a point, then querying for all records within a radius of a point using `ST_Distance` or similar functions [43]. The advantage of this route is **powerful querying and analysis** using familiar SQL, and benefiting from the maturity of relational databases (ACID transactions, etc.) [44]. We should ensure the location column is indexed (PostGIS uses R-tree indices under the hood for this [45]) so that queries like "find nearby objects" are efficient. If our backend is likely to do complex geospatial analysis (like clustering points, route calculations, etc.), PostGIS offers an extensive toolset.

- **NoSQL with Geo Capabilities:** Many NoSQL databases also support geospatial indexing. For instance, MongoDB allows storing coordinates in documents (often as GeoJSON points) and can perform geo-queries like `$near` to find documents within a certain distance of a point [46] [47]. In our context, if we store each object (or each location event) as a document with a `location` field, we could index that field as a 2dsphere index and use queries to get nearby locations or to do bounding-box searches. The **flexible schema** nature of a document store combined with these geo-queries makes MongoDB "quite suitable for geospatial applications" [48]. The example in Baeldung's geospatial app guide shows a Mongo query finding documents within 500–1000m of a given point using GeoJSON and `$near` [49], which illustrates how straightforward it can be. The performance and scaling of this approach are generally good for large volumes of location points, as MongoDB is designed to scale out horizontally and can handle high write loads (important if, say, thousands of location updates are coming in from IoT devices or mobile users).

In making a choice, we might consider the team's expertise and the expected query patterns. **If transactional integrity and complex relational queries across data are paramount**, a PostGIS-backed relational store might be preferred (especially for something like an inventory tracking app that might join location data with inventory records, etc.). **If ultimate flexibility and horizontal scale is more important**, a NoSQL store with geospatial indexing is attractive – especially since it aligns with the dynamic schema approach for custom fields.

It's also possible to adopt a **hybrid**: for example, use a relational DB for core structured data (and simpler location needs) and augment it with a specialized search or geo service (like Elasticsearch or a spatial index service) for advanced geospatial querying. But that may be overkill initially.

Either way, the backend will expose high-level operations to the application developers: e.g. an API endpoint like `/objects/nearby?lat=X&lng=Y&radius=Z` or a query to fetch all location records for an object within a date range, etc. The implementation uses the underlying geo features to fulfill these. The use cases given (tracking points on a ranch map, incidents on a neighborhood map, visited places on a globe) mostly involve point data on maps, which are well-supported by the above technologies. If we anticipate features like drawing regions (geo-fences) or computing heatmaps, we should ensure our chosen spatial solution can handle those (both PostGIS and many NoSQL can).

Additionally, **time** is an important dimension since we're tracking history. We may combine time and space queries (e.g. "incidents in the past week in this area"). This means we should also index timestamps and possibly consider time-series optimization if data becomes huge. Some databases (like Mongo or Postgres) can partition or index by time to keep recent data quick to access. We might not need a full time-series

database, but designing the schema to include a timestamp in the location records and indexing it is important for performance.

In summary, the backend's geospatial capability ensures that each app can not only store coordinates but also derive meaningful insights: **map visualizations, proximity searches, clustering of points, etc.** By using proven spatial tech (PostGIS's geometry types and functions or MongoDB's geo queries), we empower all frontends to easily build features like *"show me everything near me"* or *"plot all our travel photos on the world map"*. This heavy lifting is done once in the backend, so every app – from the smallest family travel map to the largest enterprise asset tracker – can rely on fast, accurate geolocation operations out of the box.

API and Frontend Integration

With the backend services and data model in place, we need to make it **accessible to client applications**. The design calls for allowing users (or third-party developers) to **author their own frontends** on top of our common backend. Thus, we must provide a **clean, documented API** that exposes all necessary functionality in a generic yet flexible way.

A likely approach is to implement a **RESTful API** (or GraphQL API) where each resource corresponds to our core concepts: e.g. `POST /objects` to create a new trackable object, `GET /objects/{id}` to retrieve it (including maybe its latest location or summary), `POST /objects/{id}/locations` to add a new location event, `GET /objects/{id}/locations` to list its history, and so on. Also, endpoints for queries: `GET /locations?near=x,y` or `GET /objects/search?filter=...` depending on needed features. If using GraphQL, we could allow apps to query exactly the fields they want (which might be handy given custom fields – GraphQL could fetch those as a JSON blob or specific known fields).

The **API Gateway** or service will handle authentication (ensuring the caller is allowed access) and route requests to the correct internal service (object service, location service, etc.). It will also inject the tenant context (likely derived from the auth token or subdomain) so that each request operates in the correct tenant's scope. For example, if the family travel app calls `/objects`, the backend knows to create/list objects under tenant "Family123" only.

Since the API is uniform, **one backend can serve multiple frontends concurrently** – a web app for incident mapping and a mobile app for wildlife tracking can both talk to the same endpoints, just with different credentials and getting different data. This reduces duplicated effort: any generic feature we add to the API (like the ability to filter location history by date range or to attach a photo to an object) becomes available to all clients. In some cases, we might introduce **frontend-specific optimizations**. The "Backend for Frontend" pattern suggests we could have tailor-made endpoints if, say, one app needs a combined data fetch that others don't [10]. However, we should evaluate if that's needed or if the clients can compose the data from basic endpoints. Given the commonality, likely a well-designed generic API suffices.

Developer Experience: To make it easy for app developers (who may be internal or external) to build on our platform, we'd provide clear API documentation and possibly SDKs in popular languages. If each app owner is a developer, they can use these tools to quickly integrate. If some app owners are less technical (imagine a scenario where a family without a dev background wants to use it), we might even offer a **no-code or low-code frontend template** that can be configured. For example, a simple configurable

dashboard that can be branded and set to show that tenant's data on a map, with form builders for custom fields. This, however, is a product on its own – at minimum, a robust API allows others to create the frontends they need.

Frontend Auth & Permissions: Each frontend will authenticate (perhaps using OAuth2 or API keys). A consideration is whether a user of one app can have access to another. Likely, each app has its own user base. But if we anticipate an overlap (for example, a company administrator might oversee multiple tracking apps in the platform), using a unified identity system with roles per tenant is useful. This way a single login could grant access to multiple tenant contexts. This is similar to how Microsoft's services or Google's, where one account can be part of multiple projects/organizations. Our backend should then support listing what tenants a user has access to, and switching context appropriately.

Real-Time Updates: Depending on the use case, some apps might want real-time updates of location data (e.g., an admin watching devices move on a live map). While the base solution can use polling (the client periodically calling the API), we might also incorporate WebSocket or push notification support for live data. This could be a module that subscribes to new location events and broadcasts them to interested clients in that tenant (for example, using a Pub/Sub system or something like Firebase if not building in-house). This is an advanced but valuable feature for certain domains (like live conference attendee tracking or security incident monitoring).

Customization in Frontends: Since frontends will reflect the custom fields and unique aspects of each app, they will heavily rely on the metadata from the backend. The API should provide endpoints to fetch field definitions (e.g., `GET /schema/fields?objectType=Incident` returning that tenant's custom field list and types). This allows frontends to be more dynamic and reduces hardcoding. In the Dylan Lee example, the frontend first calls for field definitions and then renders the form accordingly [36] – our approach would mirror that for any configurable portions.

Overall, the backend serves as a **unified platform-as-a-service** for location tracking: multiple frontends plug into it via well-defined APIs. This decoupling means frontends can be written in any technology (mobile apps, web apps, even command-line scripts) and as long as they follow the API contract, they will interoperate with the backend. It also means improvements on one side don't necessitate breaking the other – we could upgrade the backend internals without requiring frontend changes, and vice versa (maintaining API backward compatibility).

By **ensuring consistent API design and thorough documentation**, we make it straightforward for an administrator at a large bank or a hobbyist in a family to build or use a client app that suits their needs, all backed by the same reliable core.

Use Case Examples

Let's illustrate how different tracking scenarios map onto this common backend:

- **Wildlife Trailcam Photos:** An app for a rancher uses the platform to track wildlife sightings via trail cameras. In their tenant, they define a custom object type "Photo" with fields like *species*, *camera location ID*, and *image URL*. Each time a trail camera captures an animal, a new Tracked Object is created (or an existing animal object is updated) with a Location Record (the GPS coordinates of the

camera and timestamp). The rancher's frontend might be a web dashboard showing a map of the ranch with pins for each sighting; clicking a pin shows the photo and species info. All of this is facilitated by the core: the location data is stored and queryable (e.g., filter sightings by area), the image is stored via the media service, and the species field is simply a custom attribute in the JSON. The **mapping** and *"nearby sightings"* features come for free from our geospatial queries, and if the rancher wants to add a field (say, "animal count" if sometimes a photo has multiple deer), they can add it without fuss to their schema.

- **Conference Attendee Tracking:** A conference organizer uses a mobile app to track attendees (perhaps through badge scans or GPS if on-site). The tracked object might be "Attendee" with fields like *name*, *affiliation*, *ticket type*. Location Records might be generated when they check into a session or enter a zone (with coordinates, or just a venue location ID mapped to coordinates). The app's frontend could show organizers a real-time count of people in each room (querying location events in the last few minutes for each area). It might also show an attendee's movement history if needed (by retrieving all their location records for the day). Our backend suits this: it handles a high volume of frequent updates (if many people move around), and with spatial queries can even do fun features like "heatmap of crowd density" by aggregating points. The tenant's custom fields (affiliation, etc.) don't interfere with, say, the wildlife app's fields – each is separate in metadata. Authentication might integrate with the conference's registration system via our user service or an OAuth integration, but once authenticated, all data is scoped to that event's tenant.

- **Neighborhood Incident Reporting:** A community safety app logs incidents (like potholes, break-ins, lost pets). Here the object type is "Incident" with custom fields such as *description*, *category*, *status* (open/resolved). Every report a user submits creates an Incident object with a Location Record where it happened. The backend's geospatial abilities allow the app to implement features like *"show incidents near me"* or to send alerts for incidents within, say, 1 mile of a user's home. Because time is relevant, users could filter to *recent incidents* which is just a query by timestamp and location. The common backend makes it easy to aggregate data for dashboards too – an admin could use the API to pull all incidents in the past month and perhaps do analytics (since all data is consistently structured under the hood). If the community wants to add, say, a *priority level* field later, it's a simple addition in their configuration. And since this might involve user-generated content, our platform's user auth module plus maybe a moderation add-on can help manage who can create or edit incidents.

- **Global Travel Log (Places Visited):** A family app marks all the places they've traveled together. Each object might be "VisitedPlace" with fields like *place name*, *date*, and maybe *photo album link*. When they add a new pin to their world map, the app creates an object with the coordinates and date. Over time they have a collection of Location Records (though for this use case, maybe each place is a single record – or if they revisit, multiple records). The backend stores all those points and allows the app to simply fetch all of them to display on a globe. If they want to see the sequence of their journey, they can sort by date (supported by our data model easily). Even though this is a small-scale usage (just one family), they benefit from the industrial-strength backend – their data is safely stored, and they can access it from multiple devices. The multi-tenant design doesn't add much overhead for them, it just isolates their data (they're essentially one tenant). They likely don't need super custom fields beyond maybe a note or rating – but if they do, they can add it.

These examples show that, despite differing in domain, they all follow a pattern: define object type and fields -> record locations -> use core features to retrieve or analyze those records. The **common platform** serves as the engine for all, while the **frontends and configurations** give each app its unique personality.

Scalability and Performance Considerations

A crucial advantage of a shared backend is that we can invest in making it **highly scalable and performant** for all clients. Here are some considerations to ensure the system can handle a wide range of loads, from a single-family's lightweight usage up to an enterprise with thousands of objects and continuous tracking:

- **Efficient Data Access:** We will create appropriate indexes on the database for key fields such as Tenant ID, Object ID, and the spatial index on locations. For example, in a SQL-based store, an index on `(TenantID, ObjectID)` for the objects table, and a spatial index on the `location` field in the locations table (possibly a composite index with TenantID as well, to quickly filter by tenant then location). This ensures that queries like "get all objects for tenant X" or "find locations near Y for tenant X" are fast. In a NoSQL scenario, partitioning by tenant or using separate collections per tenant can achieve similar isolation and performance (most queries would automatically only scan that tenant's data). Many multi-tenant systems add the tenant identifier as a prefix to keys or partition key (e.g., in Cassandra or DynamoDB, a partition key could be tenantID) to distribute load and keep each tenant's data group accessible together.

- **Horizontal Scalability:** The modular design allows us to scale services independently. For instance, if the Location Tracking service is heavily used (lots of writes/reads of coordinates), we can run multiple instances of it behind a load balancer, or scale up the database resources for that part (like using a cluster of database nodes). Cloud infrastructure makes this easier – deploying on AWS, Azure, or GCP, we could use managed database services (Aurora, CosmosDB, etc. depending on SQL/NoSQL choice) that automatically handle replication and scaling. The stateless API servers can be containerized and scaled out as demand grows. We might also leverage caching layers (e.g., Redis) to store frequently accessed data or expensive query results (for example, the list of all objects for a tenant could be cached since it doesn't change often, or spatial query results could be cached for a short time if many users ask similar queries). This ensures that as more users or more devices come onto the platform, the performance remains snappy.

- **Large Data Volume Management:** Some apps, like a bank inventory or fleet tracking, might generate huge volumes of location data (imagine tracking 10,000 vehicles, one update per second – that's a lot of records!). For such scale, we might incorporate **data lifecycle management**: partition older data to cheaper storage, provide aggregation (so queries can fetch summarized data for older periods), and so forth. The design can include an archiving job per tenant that moves very old location records to an archive (or deletes them if not needed) to keep the working set smaller. Additionally, if using SQL, features like partitioned tables by date can keep inserts and queries efficient by pruning partitions.

- **Reliability and Fault Tolerance:** Consolidating multiple apps on one backend means a failure affects many users, so we must design for high availability. This means redundant instances of each service (no single points of failure), use of backups for databases, and possibly multi-region deployment if global users need low latency or redundancy. If one tenant suddenly causes a spike (say an organization runs a big event causing a flood of location updates), our system should handle

it or degrade gracefully (throttling that tenant if necessary) rather than crashing entirely. Isolation can also play a role here – for extremely large tenants, we might isolate them on their own resources (as discussed) so they don't degrade others.

- **Testing at Scale:** Because the platform could be used in scenarios we didn't even imagine yet, it's important to test with varied data shapes and loads. We should simulate usage like "100 tenants each with 1000 objects" as well as "1 tenant with 1,000,000 objects" to see how the system behaves. Optimizations like query tuning or additional indexes might emerge from this. We might find, for example, that some queries need to be materialized (precomputed) for performance – e.g., keeping track of an object's latest location in the object table to avoid always joining to the locations table for simple queries. Such refinements will make the system fast for both small and large tenants.

- **Monitoring and Analytics:** A common backend allows us to centrally monitor performance metrics and usage patterns. We can identify which features are heavily used and optimize them, and also detect any abuse or anomaly (like if an automated client is hitting the API too hard). Having good logging per tenant also ensures that if one app has an issue, we can pinpoint it. This is especially important if we offer this as a service to others – we'd need to meet certain SLAs. Cloud monitoring tools or custom dashboards can be set up to watch query latency, error rates, etc. across the multi-tenant system.

- **Security & Compliance:** On the performance side, using encryption (at rest or in transit) is a slight overhead but usually mandatory for sensitive data. We will encrypt data in the database as needed (making careful choices, since encrypting certain fields might mean they can't be indexed [50] – often latitude/longitude are not encrypted so that spatial index can work, but personal data could be). We also ensure that proper secure coding practices are followed since one breach could expose all tenants. Regular audits and perhaps even allowing tenants to have their own encryption keys (for very sensitive use cases) could be considered in advanced scenarios.

By addressing these considerations, the backend will be **robust and scalable by design**. The fact that the architecture is modular also means we can improve components without affecting the whole system. For example, if we need more throughput for location writes, we could introduce a message queue between the API and the database (so writes are buffered), or use an alternative high-throughput data store for raw location logging (like a time-series DB) that then feeds into the main DB for querying. The modular approach and multi-tenant awareness ensure we have the flexibility to make such changes per need. In the end, whether it's 5 users or 5 million, the system should gracefully scale to handle the load.

Conclusion

By developing a **common modular backend for location tracking**, we solve the core problem once and empower a multitude of applications to build on top of it. This approach yields a number of compelling benefits:

- **Reusability and Speed of Development:** New tracking apps can be launched quickly by reusing the proven backend. Developers (or even "citizen developers") only need to create the frontend and define any custom data fields; the heavy backend functionality is already there. As noted in the GIS SaaS context, many solutions need the same core features, and having a shared platform means you

"build once, sell (or use) many times" [1]. Our system becomes a foundation for innovation in any domain where location matters.

- **Consistency and Reliability:** All apps share the same data structures and services, which means data is managed in a consistent way. Bugs are fewer (since we fix them centrally), and data quality is higher. A common data model across applications also enables **integration and analysis** across them (if ever needed) without a lot of transformation. It's akin to having a standard – as long as everything conforms to it, things work together smoothly [2]. If the family travel app and the conference app both chose to share data (imagine a scenario where one could overlay incident data on travel maps), it'd be feasible thanks to the unified model.

- **Customization and Flexibility:** Despite the single core, each app feels fully custom. Through the extensibility mechanisms, app owners can tailor the data model to their needs without affecting others [38]. They can also brand their frontends and have unique user experiences, but all backed by a stable service. The platform supports this with metadata-driven configurability, proving that flexibility doesn't have to come at the cost of maintainability.

- **Scalability and Efficiency:** Instead of each app team worrying about scaling their own backend, our centralized backend team ensures the platform scales. Resources (servers, databases) are utilized more efficiently by pooling for multi-tenancy, which lowers costs per app especially for smaller ones [13]. If one app never grows big, it simply uses a small slice of the shared resources; if another becomes a hit, the shared infrastructure can dynamically give it more without a full redesign. This elasticity benefits everyone.

- **Simplified Maintenance:** Upgrading one system is easier than coordinating many. Whether it's a security patch or a new feature (say, adding support for a new map provider or a new kind of query), we do it once. We can also instrument and monitor one system. From the user perspective (the app owners), they get improvements continuously without needing to hire backend specialists – they effectively outsource the backend development to this common platform.

In solving the problem, we've essentially outlined a **multi-tenant, location-tracking backend platform**. This approach is aligned with modern cloud SaaS design and leverages best practices from industry (like shared Common Data Models, microservice architecture, and dynamic schemas for customization). By focusing on the shared problem of tracking "objects" over space and time, we ensure the system is abstract enough to serve *diverse scenarios*, yet concrete enough to be efficient and reliable in each.

In conclusion, the creation of a common modular backend not only addresses the initial problem (avoiding duplicate efforts in building similar apps), but it also opens the door for a scalable service that could potentially be offered to others with similar needs. From a lone family mapping their adventures to an enterprise monitoring assets worldwide, all can benefit from the **unified, robust, and adaptable** infrastructure we put in place. This unity in the backend paired with diversity in the frontend is a powerful model for software product lines going forward, enabling rapid development and strong user-focused solutions all at once.

Sources:

- Microsoft Common Data Model – concept of a shared data language and extensible schemas for unified data across apps [2] [5].
- Citus Data on Multi-Tenant Architecture – discusses shared schema approach (tenant ID per row) and need for tenant isolation vs. cost trade-offs [15], as well as patterns to allow custom data per tenant without affecting others [38].
- Stacy Mwangi, *White-Label GIS SaaS* – introduction of building one core to serve many branded solutions (build once, reuse for many clients) [1], highlighting identical core needs across customers.
- Dylan Lee, *Dynamic Fields in Multi-Tenant SaaS* – describes the challenge of tenant-specific fields in an inventory system and a solution using MongoDB with metadata for flexible schemas [24] [33].
- Baeldung Geospatial App Architecture – outlines using MongoDB for geospatial queries with a flexible JSON schema [51], and using PostGIS in PostgreSQL for spatial data with SQL functions and indexes [41] [52].
- Ken Whitesell (Django Forum) – advice on multi-app backends with shared authentication/authorization, illustrating how one core can manage access to multiple app-specific areas [17] [53].

---

[1] [21] Python White-Label GIS SaaS: Build Multi-Tenant Geospatial Platforms That Scale to Thousands of... | by Stacy Mwangi | Oct, 2025 | Medium
https://medium.com/@stacyfuende/python-white-label-gis-saas-build-multi-tenant-geospatial-platforms-that-scale-to-thousands-of-6c78e6513b49

[2] [4] [5] [6] Common Data Model - Common Data Model | Microsoft Learn
https://learn.microsoft.com/en-us/common-data-model/

[3] [13] [14] [15] [16] [18] [23] [29] [30] [38] [39] [40] [50] docs.citusdata.com
https://docs.citusdata.com/en/stable/_static/mt-data-arch.pdf

[7] [8] [17] [53] Advice for building an app backend to host multiple apps in the future - Getting Started - Django Forum
https://forum.djangoproject.com/t/advice-for-building-an-app-backend-to-host-multiple-apps-in-the-future/20283

[9] [10] Backends for Frontends Pattern - Azure Architecture Center
https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends

[11] [12] Scaling Our SaaS: Building a Unified Multi-App Architecture with Shared Components | by Tarikul Islam | Dec, 2025 | Medium
https://medium.com/@tituhinkh/scaling-our-saas-building-a-unified-multi-app-architecture-with-shared-components-efda9962cba3

[19] [20] [22] Tenancy Models for a Multitenant Solution - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/considerations/tenancy-models

[24] [25] [26] [27] [28] [33] [34] [35] [36] [37] A Smarter Way to Handle Dynamic Fields in Multi-Tenant SaaS | by Dylan Lee | Medium
https://medium.com/@inexpressible2510/a-smarter-way-to-handle-dynamic-fields-in-multi-tenant-saas-369395c41d43

[31] [32] Multi Tenancy - Extensible Data Model - Ayende @ Rahien
https://www.ayende.com/blog/3498/multi-tenancy-extensible-data-model

41 42 43 44 45 46 47 48 49 51 52 Architecture of a Geospatial Application with Java | Baeldung
https://www.baeldung.com/java-geospatial-applications