Modeling Data in TrueVault

Fundamentals

TrueVault is a highly secure, highly available, and highly durable distributed data store. Hosted securely in the cloud, you get all the benefits of a complex distributed system with none of the operational burden. Simply use the TrueVault REST API to store and retrieve data, and we take care of the rest. You can update your data model dynamically, and selectively index the documents and fields you want to search against.

TrueVault stores your data securely in multiple storage engines to optimize the most common access patterns. The details are abstracted behind our API, so the complexity is hidden. From a data modeling perspective, think of TrueVault as a distributed, NoSQL document store. Simply send your JSON documents to the TrueVault API, and we'll handle the encryption, key management and redundant storage.

In Context

The TrueVault secure BaaS is unique, but you can use the same data modeling techniques you would use with:

MongoDB
DocumentDB
RavenDB

```
{
  “first_name”: “John”,
  “last_name”: “Smith”,
  “phone”: “555.555.5555”,
  “primary_care_nickname”: “Dr. Joh
  “primary_care_id”: “522090eb-0e60-
  “allergies” : [{
       “type”: “FOOD”
       “trigger”: {
          “name”: “peanut”,
          “group”: “legume”
          “exposure_type”: “INGEST”
       },
       “reaction”: {
         “severity”: “CRITICAL”,
         “type”: “ANAPHYLAXIS”,
         “treatment_window”: “5m”
       }
    },{
       “type”: “ENVIRONMENT”,
       “trigger”: {
         “name”: “grass”,
         “exposure_type”: “CONTACT”
       },
       “reaction”:: {
         “severity”: “MODERATE”,
         “type”: “HIVES”,
```

Design for Access

We recommend you model your data in TrueVault based on your access patterns, not based on the inherent relationships in your data. This often means that you need to denormalize parts of your data model to satisfy your specific query patterns. That is, you may store some data multiple times to simplify your queries.

Be judicious in your denormalization decisions. Duplicate information only where the improved query efficiency outweighs the effort of maintaining two copies.

In the excerpt to the left, this patient record denormalizes a nickname for their primary care physician while the details for that physician are stored in a separate document. It also stores all of the patient's allergy information embedded within the patient document. This structure would allow a rich patient profile view to be built from a single document request, which means an informative patient list-view could be driven from a single search. If you plan to preview records in a list view, this approach is crucial. The alternative, fetching a doctor or allergy by id for each patient in the list, results in the dreaded *n+1 requests problem* and could make your application feel sluggish.

The embedded allergy structure is not inherently good or bad; it depends entirely on the access patterns these data experience. If a patient's allergies are only ever displayed in the context of that patient, this approach is ideal. One request loads all the data you need. On the other hand, if you plan to browse allergies first, then find the patients with those allergies, this model would be inefficient. In that case, normalizing the data would be more appropriate.

What if you need to support both access patterns? In that case, store the data twice, making each access efficient. Don't worry about the data bloat, TrueVault can handle it!