Savvy ZooKeeper

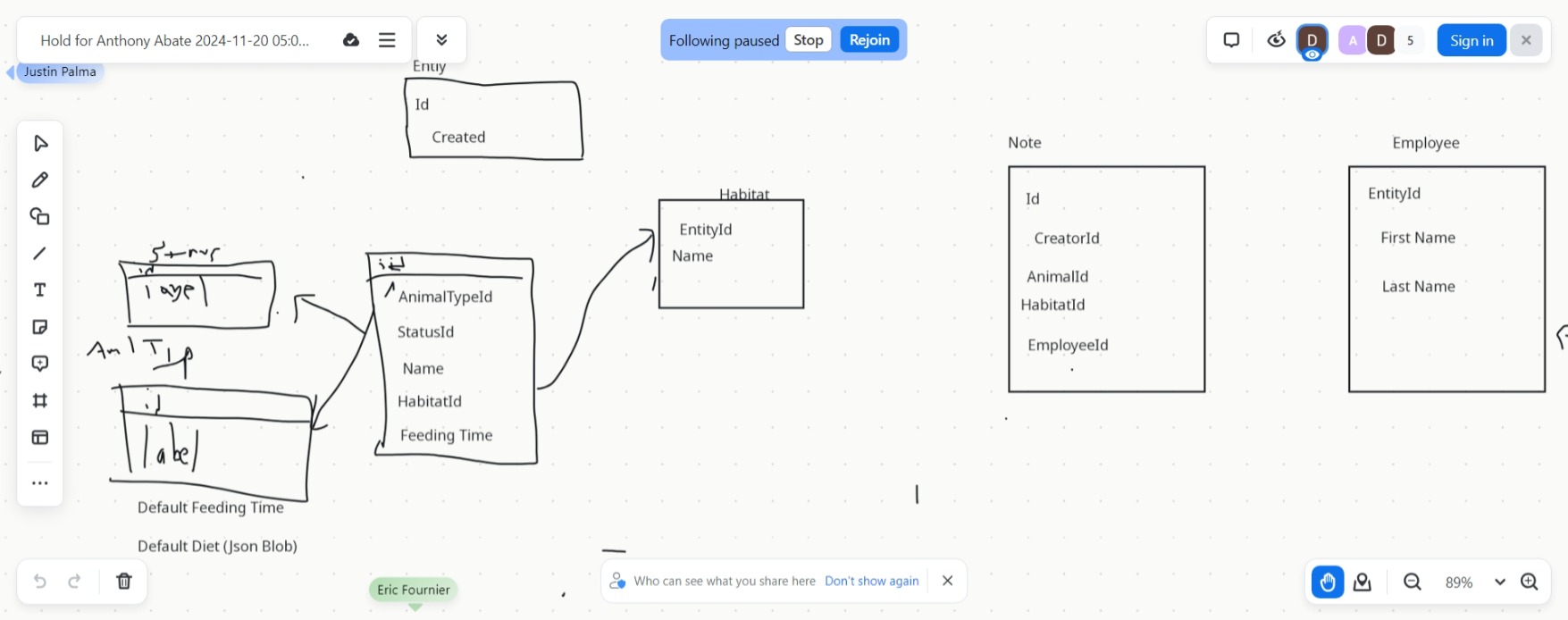
(I know there is an Apache Zookeeper, but I couldn’t think of a better name)

**Important:** In an effort to produce a reasonable amount of code in a reasonable amount of time there are some caveats and things to keep in mind:

1. The data model is complete with FKs, Indexes and constraints, where appropriate.
2. The REST API is mostly a **skeleton** – except where required to implement APIs similar to the ones required for the project demonstrations (see API section for endpoint mapping)
3. Much of the UI is **read-only** but allows navigation of data created via the REST endpoints

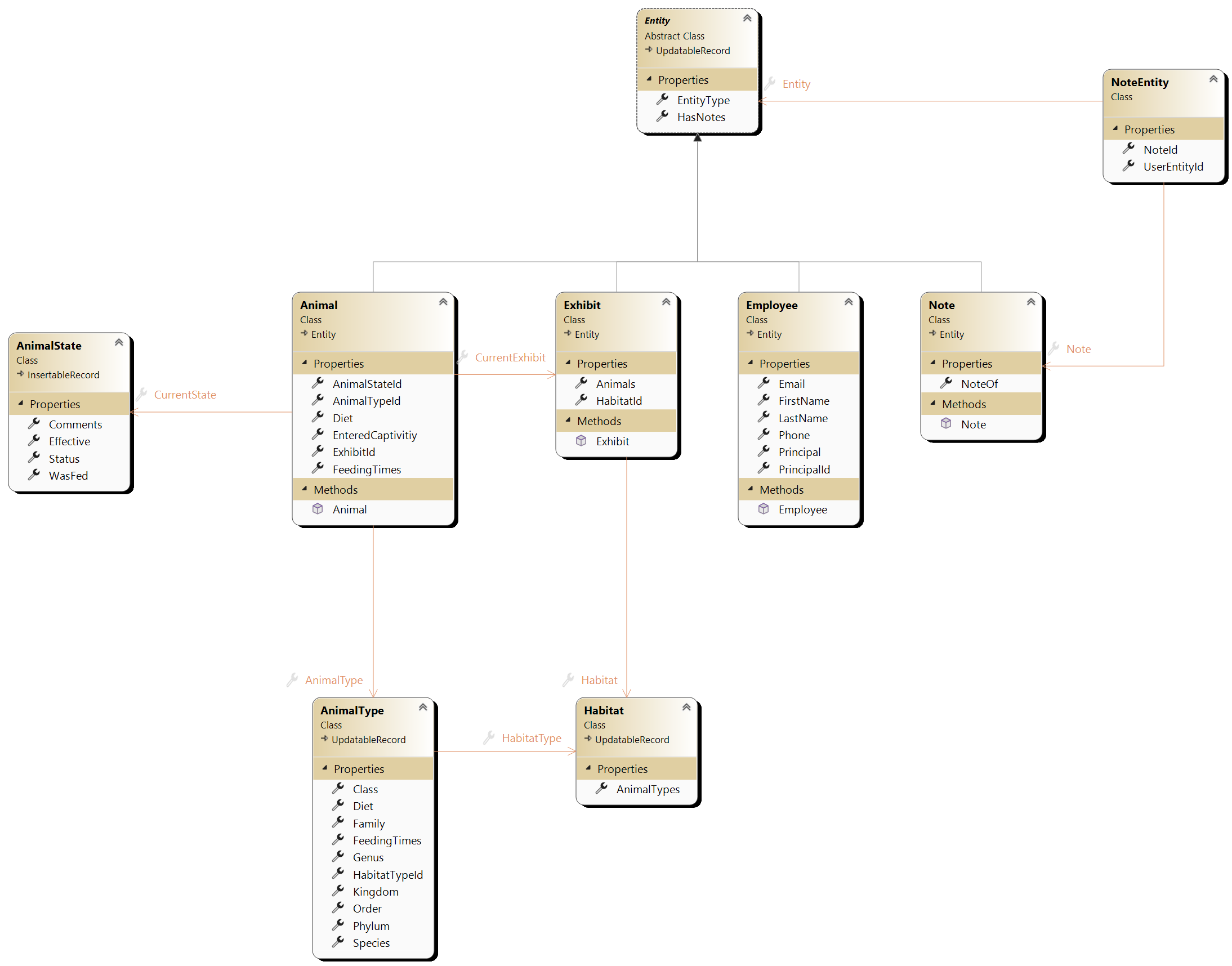
# Data Model

First task was to come up with a data model that represented all the requirements.



The original design morphed into 3 main data schemas: Metadata, Data and Security

## Data (Entity) Tables



These tables are client specific data. Employees will create and interact with records in these tables.

These tables also follow a TPT (Table-Per-Type) hierarchy with a base table **Entity**

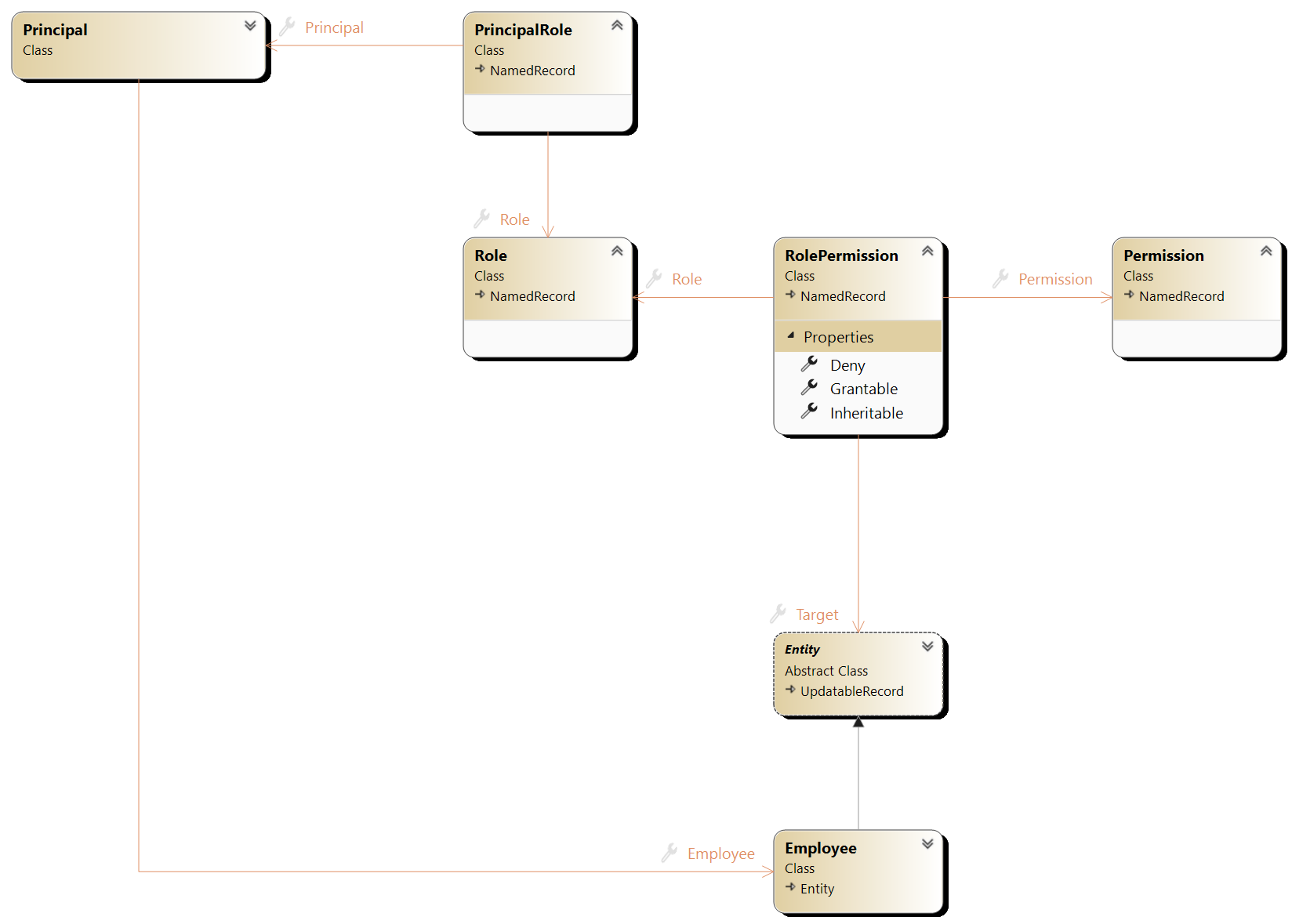
**NoteEntity** is a many-to-many table allowing the same note to be assigned to any entity in the TPT hierarchy. Since a Note is also an Entity, Notes-Notes is actually allowed by the schema, but could be disabled via business rules or an sql check constraint. Alternatively, Notes could be split out of the TPT hierarchy.

**AnimalState** is an ‘insert only’ table containing the status of animals as a historical time series.

## Metadata Tables

This data that is generally *static* and is supplied by the application. In our application, the **AnimalType** and **Habitat** data falls into this category. Barring unusual circumstances, species data will not change nor will customers add or be expected to maintain this dataset in application. A ‘metadata/reference data’ administrator would be responsible for these tables. Client specific overrides would be handled in other tables, but metadata should be a ‘golden’ dataset.

## Security Tables



Security tables are all related to permissions, roles, role membership

Some take aways from the security data model:

Every record in the system needs a valid principal for tracking create and/or update operations.

A principal need not be associated to an employee. This is the case for the system account that creates the default data. Roles are bundles of permissions. Permissions can have an optional target entity. This means a permission of ‘read’ can be constrained to specific entity (or subtree if inherit flag is set)

## Database Transactions and Concurrency Conflicts

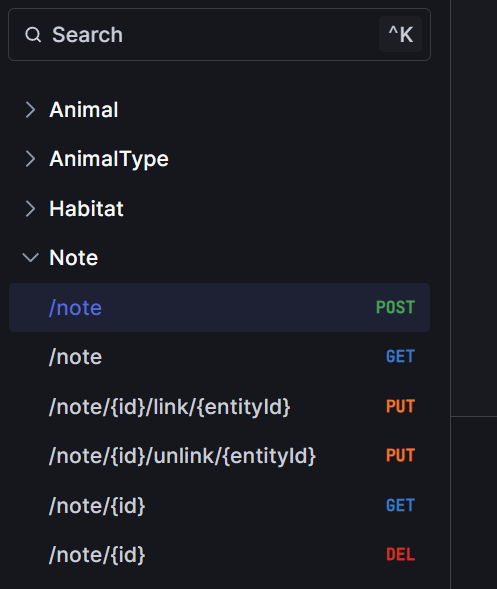
1. There is no need to explicitly create or manage transactions using the ORM entity framework – it provides transactions out of the box: [Transactions - EF Core | Microsoft Learn](https://learn.microsoft.com/en-us/ef/core/saving/transactions#default-transaction-behavior)
2. I enabled optimistic concurrency for all updatable entities by adding a special concurrency token natively supported by the underling datastore: [Handling Concurrency Conflicts - EF Core | Microsoft Learn](https://learn.microsoft.com/en-us/ef/core/saving/concurrency?tabs=data-annotations#optimistic-concurrency)

## API Implementation

Instead of GraphQL (which uses node.js) equivalent / similar rest APIs were created which can be browsed using a builtin Scalar API endpoint viewer (<https://scalar.com/>)

**Why didn’t I use GraphQL?** GraphQL can probably help with some of the over/under posting of extra REST/HTTP requests, but getting up and running + integrated seemed a bit daunting in the context of coding assessment.

Each REST endpoint is broken up into nouns and then appropriate verbs:



To handle the **createNote** endpoint one can:

1. **POST** a new note **PUT** to link to the appropriate entities
2. **POST** a new note with the entities field pre-populated