Savvy ZooKeeper

(I know there is an Apache project called Zookeeper, but I couldn’t think of a better name)  
Website: [Zoo Keeper Dashboard](https://savvyzookeeper20241202161819.azurewebsites.net/)  
Source Code: [abbotware/savvy](https://github.com/abbotware/savvy)

Folders Overview  
docs\ - document related files (including this document)

csharp\Savvy.ZooKeeper.Models\ – Database Model   
csharp\Savvy.ZooKeeper\ – Main Application  
csharp\Savvy.ZooKeeper.Tests\ – Example unit test with mock / in memory database

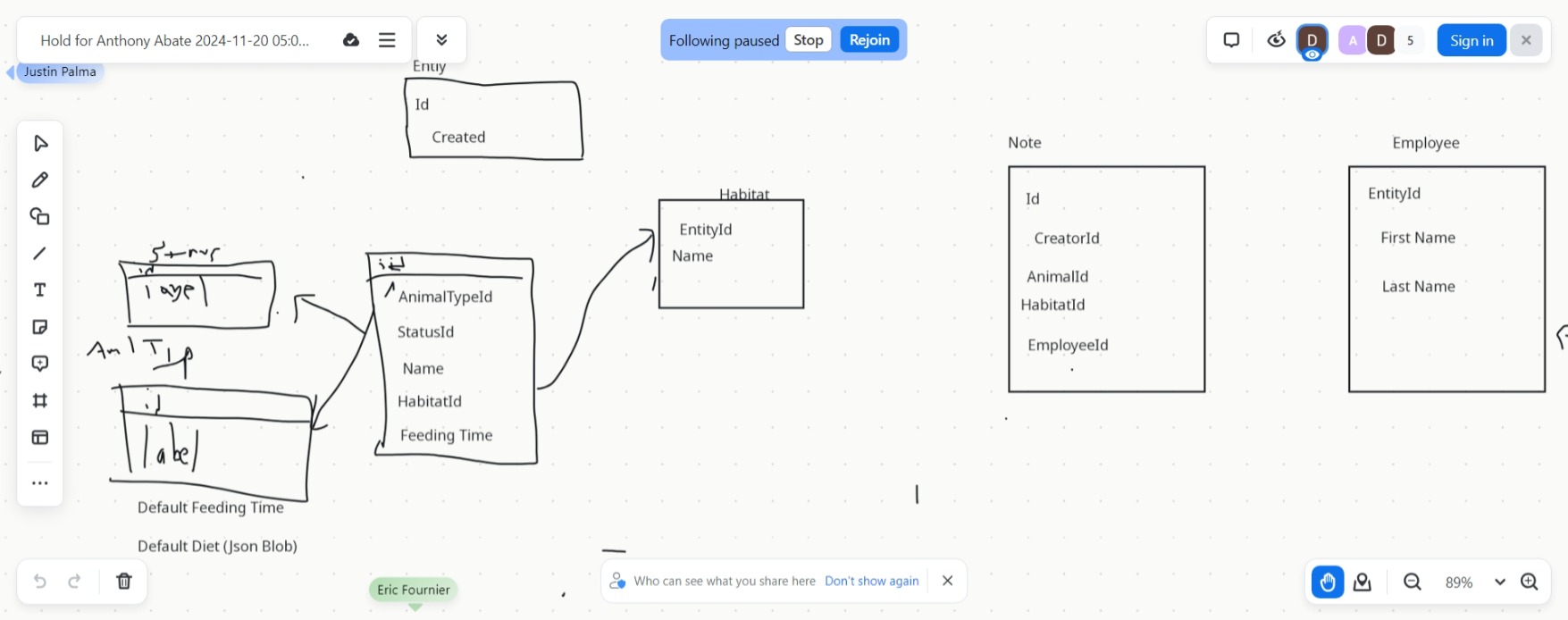
csharp\Savvy.Aspire.AppHost (used for container orchestration)  
csharp\Savvy.Aspire.ServiceDefaults (used for container orchestration)

**Important:** In an effort to produce a reasonable amount of working 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

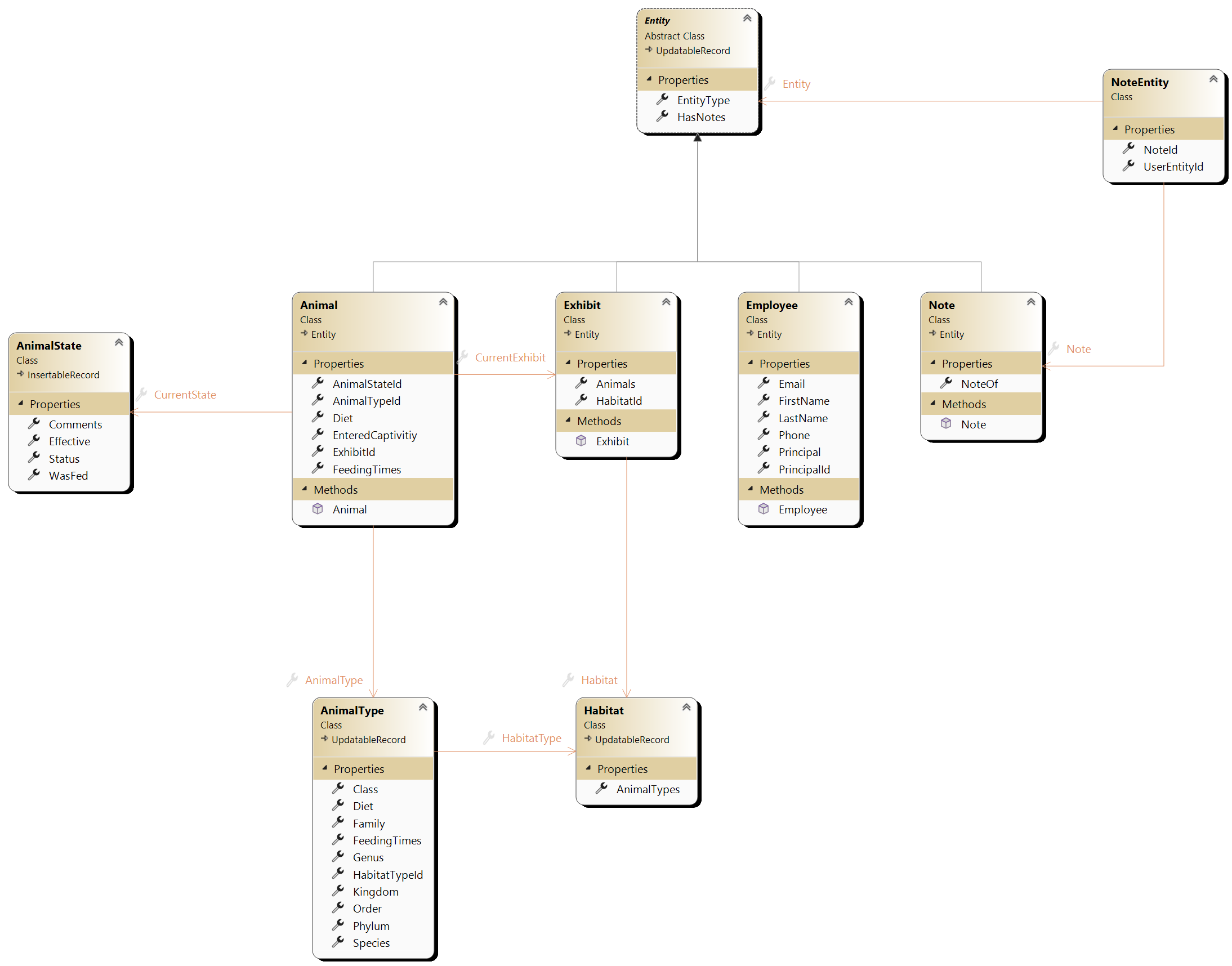
# Data Model

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



The original design morphed into 3 main data schemas: **Data**, **Metadata**, and **Security.** Browsingthe files in the Savvy.ZooKeeper.Models it should be straightforward. C# allows the use of attributes to decorate classes for the runtime. There are both GUI and Storage attributes which mean that during rapid development, the Database models can be the same as the models used in the API and GUI layers. In the future these could be split out so they may vary independently to version control Public API definitions. They are currently 1:1 with only a few exceptions: Some create methods (ie to create the animal and exhibit at the same time) require specialized ‘create’ models.

## Data (Entity) Tables



**Entity Tables**

**Metadata Tables**

The **Entity** 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**

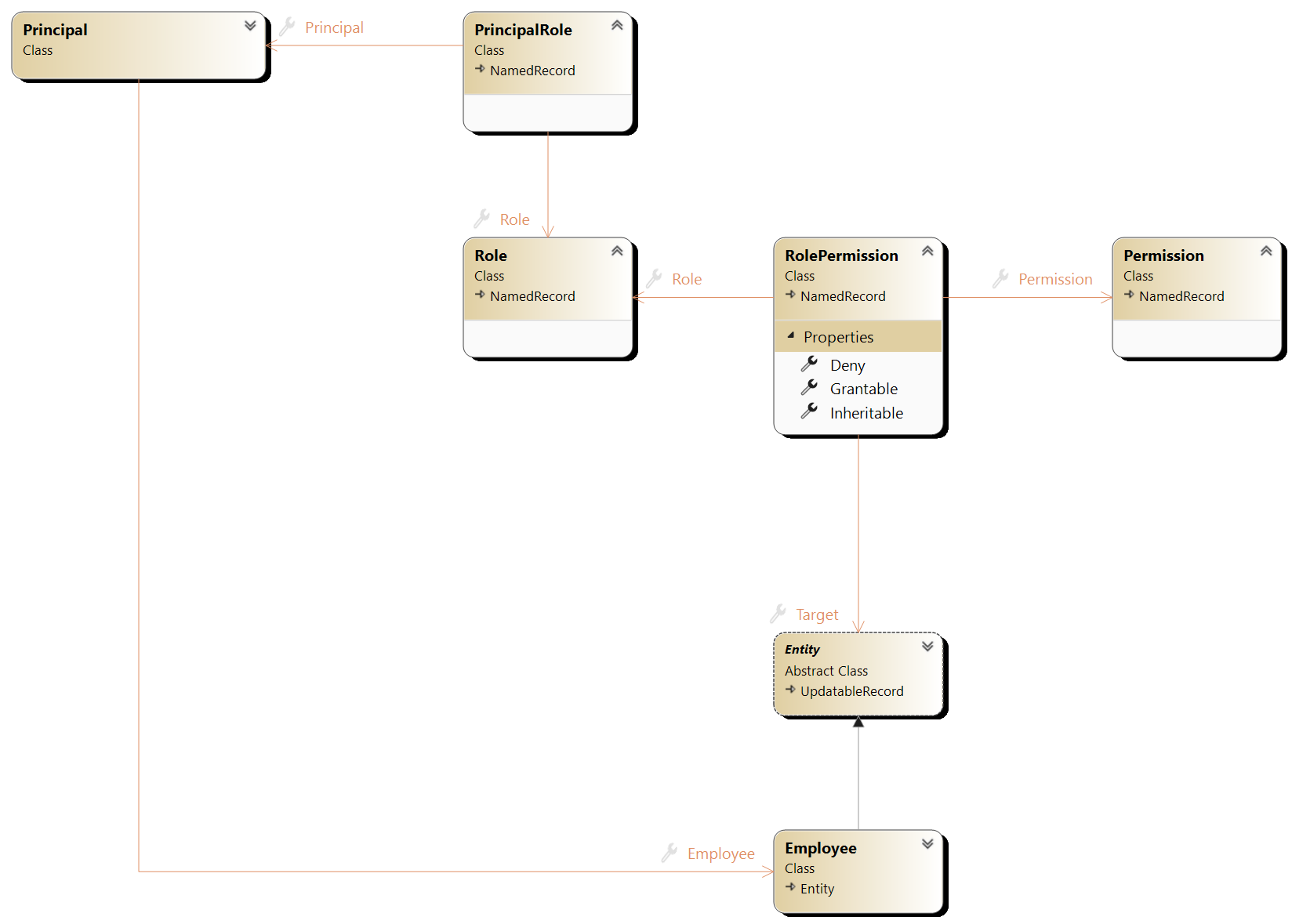
**NoteEntity** is a many-to-many table allowing the same note to be assigned to any entity in the TPT hierarchy. Since a Note is also an Entity, Notes-Notes is actually allowed by the schema, but could be disabled via business rules or an sql check constraint. Alternatively, Notes could be split out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

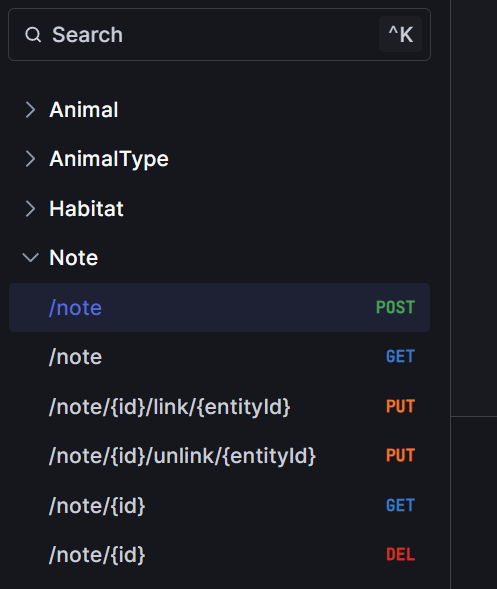
## Database Transactions and Concurrency Conflicts

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, I opted for equivalent / similar REST APIs which can be browsed using a built-in Scalar API endpoint viewer (<https://scalar.com/>)

Each REST endpoint is broken up into nouns and then appropriate HTTP verbs. Except for the 3 in the requirements, not all the endpoints have been properly vetted but are at least partially functioning.

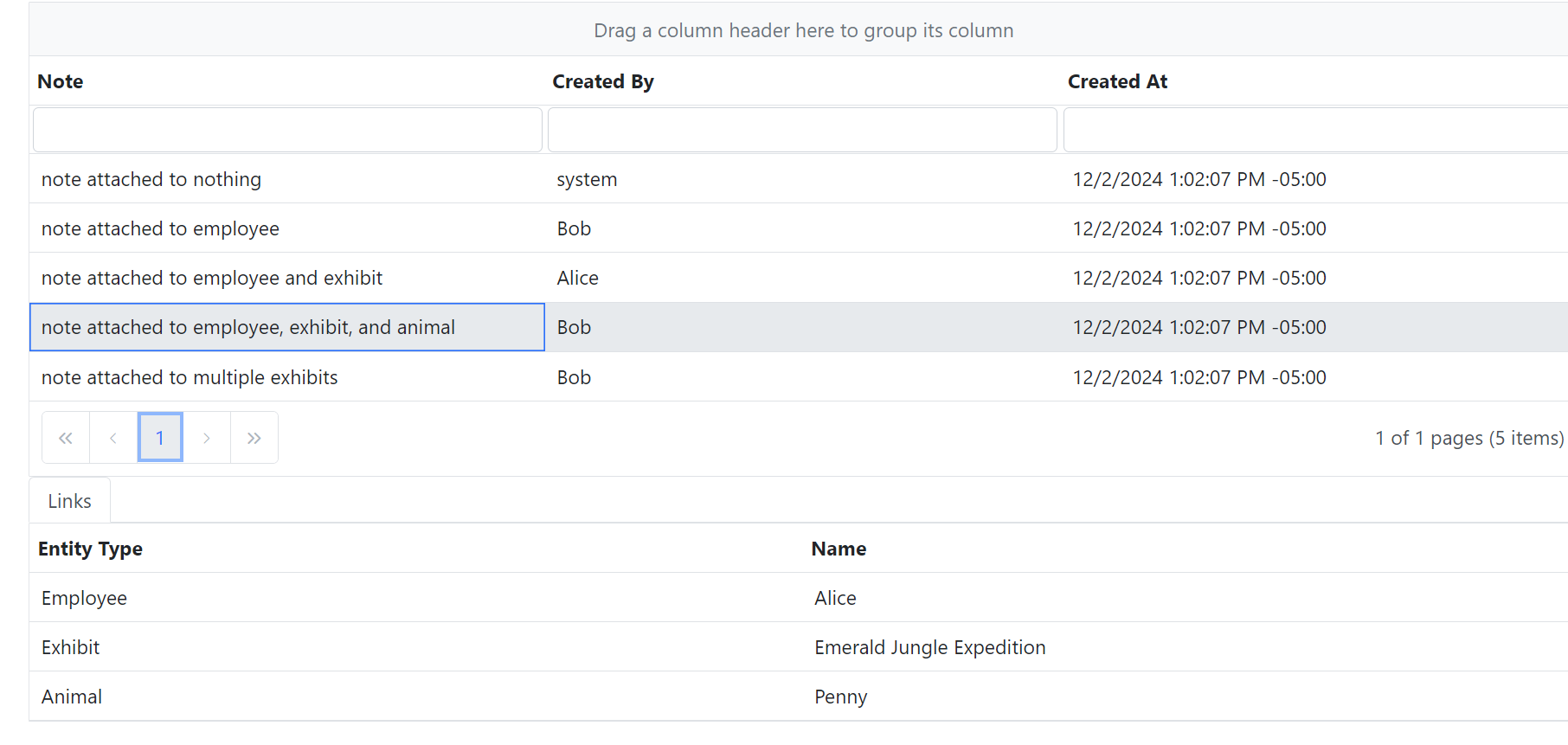


### createNote (/note POST)

To handle the **createNote** endpoint a user can do 2 things:

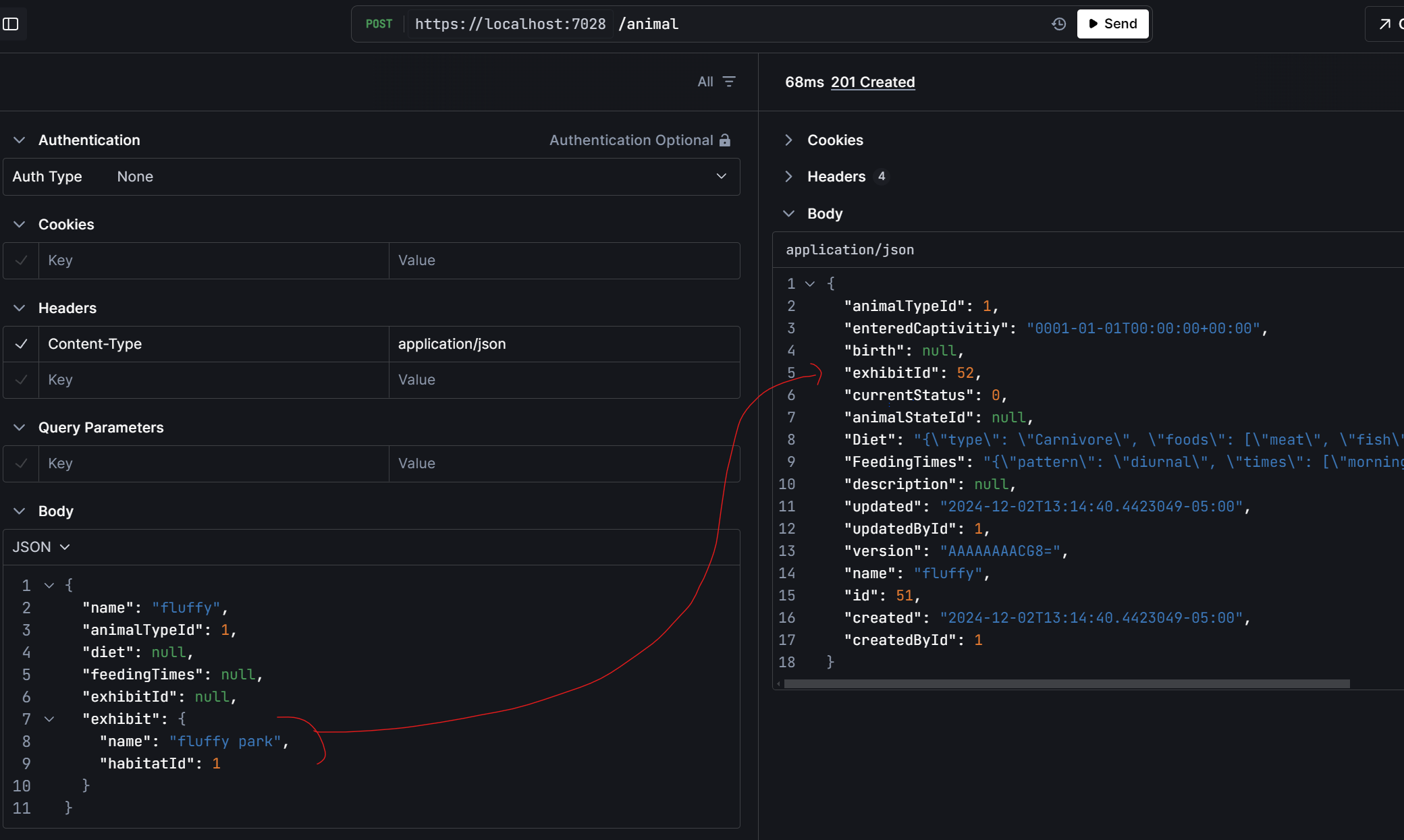
1. **POST** a new note **PUT** to link to the appropriate entities
2. **POST** a new note with the entities field pre-populated
   1. This will result in a BadRequest if the entities are not found

Here is the GUI showing a single note linked to an employee, exhibit, and animal:



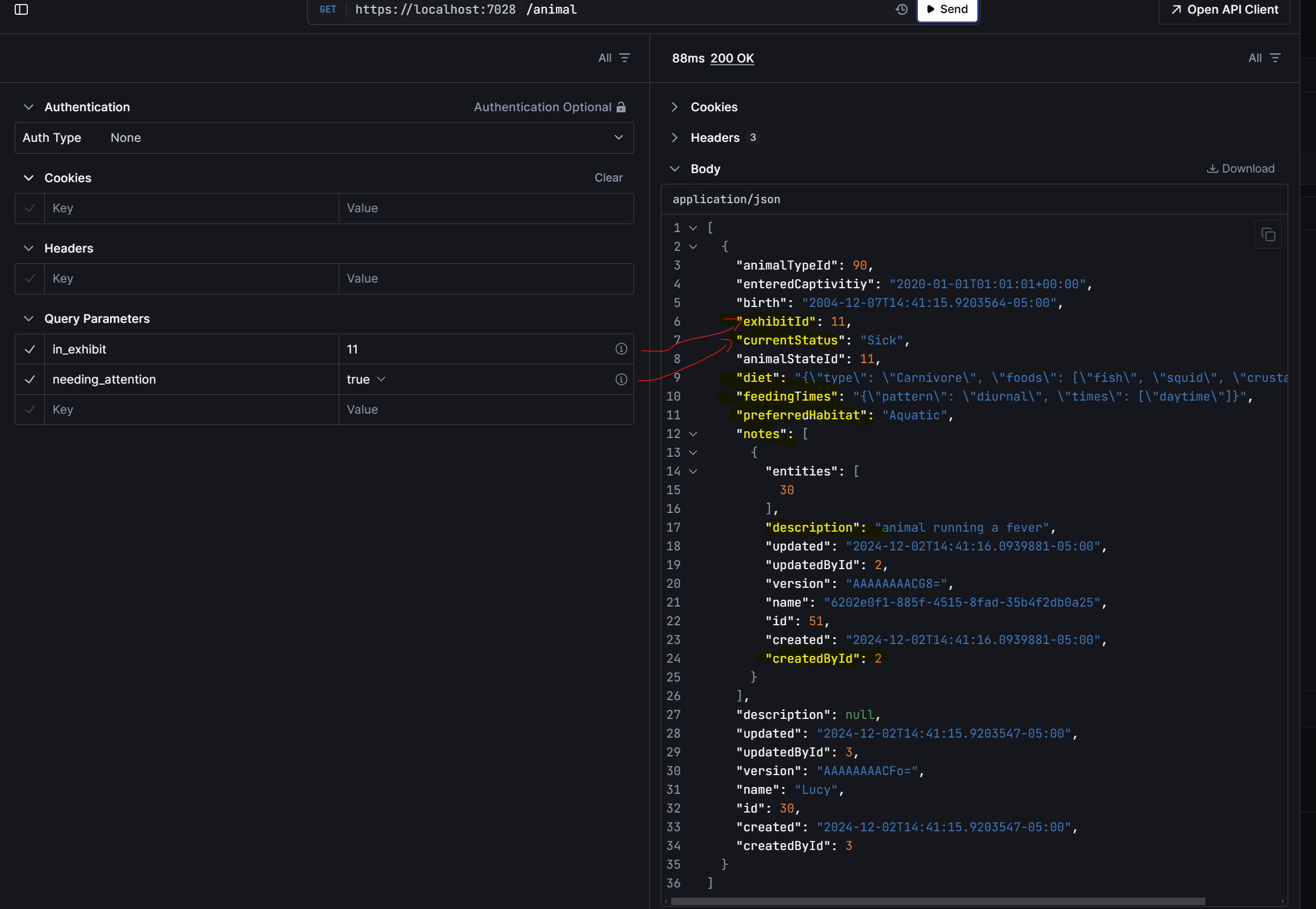
### createAnimal (/animal POST)

Create Animal can take an optional **ExhibitId** or **Exhibit** (to create a new exhibit)  
example of the post / response. Validations will reject if both are set.



### getAnimals (/animal GET)

The get animal method returns all animals or can be filtered – below both filters are applied highlighting the values specified in the API requirements.



## PII Data

The PII data has been masked so that an employee that does not have the view PII permission cannot see this data:  
  
