04 Nested Models in Pydantic

Nested Models

Introduction to Nested Models

When modeling complex data structures, we often need to represent relationships between different entities. Pydantic, a data validation library for Python, provides elegant ways to create nested models that reflect these relationships. This approach allows us to build sophisticated data schemas with proper type validation and clear structure.

Basic List Fields

Let's start with adding a simple collection field to a model. Consider a `Product` model that needs to store multiple tags:

```
class Product(BaseModel):
    name: str
    price: int = Field( ... , gt=0, title="Price of the item", description="Price
must be greater than zero")
    discount: int
    discounted_price: int
    tags: list = []
```

While this works, it has a significant limitation: we haven't specified what type of elements the list should contain. This means any type of data could be added to the tags list, which isn't ideal for validation.

Typed Collections with Python's typing Module

To address this limitation, we use Python's `typing` module, which allows us to specify the element types within collections:

```
from typing import List

class Product(BaseModel):
    name: str
    price: int = Field( ... , gt=0, title="Price of the item", description="Price
must be greater than zero")
    discount: int
    discounted_price: int
    tags: List[str] = []
```

Now our model explicitly specifies that `tags` should be a list of strings. This enables Pydantic to validate that each element conforms to the expected type.

Using Sets for Unique Values

When working with collections like tags, we often want to ensure uniqueness. Lists allow duplicates, but sets enforce uniqueness automatically:

```
from typing import Set

class Product(BaseModel):
    name: str
    price: int = Field( ... , gt=0, title="Price of the item", description="Price
must be greater than zero")
    discount: int
    discounted_price: int
    tags: Set[str] = set()
```

By using `Set[str]`, we ensure that each tag is unique within the collection.

Creating Nested Models

Now let's explore true model nesting. Just as each attribute in a Pydantic model has a type like `str` or `int`, an attribute can also be another Pydantic model:

```
class Image(BaseModel):
    url: str
    name: str

class Product(BaseModel):
    name: str
    price: int = Field( ... , gt=0, title="Price of the item", description="Price
must be greater than zero")
    discount: int
    discounted_price: int
    tags: List[str] = []
    image: Image
```

This structure indicates that each product has an associated image with its own properties. When using FastAPI, the framework automatically generates appropriate request body schemas that include these nested structures.

Special Types for Enhanced Validation

Pydantic offers specialized types that inherit from basic types but provide additional validation. For example, instead of using a plain `str` for URLs, we can use `HttpUrl`:

```
from pydantic import HttpUrl

class Image(BaseModel):
    url: HttpUrl  # Will validate that this is a proper HTTP URL
    name: str
```

The `HttpUrl` type ensures that the URL string is properly formatted according to HTTP standards.

Lists of Nested Models

We can combine collections with nested models. For instance, a product might have multiple images:

```
class Product(BaseModel):
    name: str
```

```
    price: int = Field( ... , gt=0, title="Price of the item", description="Price
must be greater than zero")
    discount: int
    discounted_price: int
    tags: List[str] = []
    images: List[Image]  # A list of Image models
```

This structure allows us to associate multiple images with each product, each having its own URL and name.

Deep Nesting for Complex Relationships

Models can be nested to arbitrary depths to represent complex relationships. Let's create an `Offer` model that contains multiple products:

```
class Offer(BaseModel):
    name: str
    description: str
    price: float
    products: List[Product]
```

Now we have a three-level deep structure:

- `Offer` contains multiple `Product` instances
- Each `Product` contains multiple `Image` instances
- Each `Image` has its own properties

This demonstrates how Pydantic allows us to build rich data models that accurately represent the structure of our domain.

Key Benefits of Nested Models

1. **Structured Validation**: Validation happens at all levels of the nested structure
2. **Clear Organization**: The code clearly represents relationships between entities
3. **Automatic Documentation**: Tools like FastAPI use these models to generate schema documentation
4. **Type Safety**: The Python type system helps catch errors during development

Conclusion

Nested models in Pydantic provide a powerful way to represent complex data structures with proper validation. By combining basic types, collections, and custom models, we can build sophisticated schemas that accurately model our domain while maintaining strong typing and validation throughout.