# Nested Models

You can create a model and then nest it inside another pydantic model.

We can also nest a Python data type as a list inside other model.

Let's first take an example of adding a list as a field to other model.

Let's add a list called tags to a product model

Here is the code:

```
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 =[]
```

This creates a list called tags, however this does not allow us to declare the type of each element of the list.

To associate a type, we use List from Python's typing module.

Replace the above code with the following:

```
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 in a list, the problem is that tags might repeat themselves. Hence it is better to use a set instead.

Hence replace above code with this:

```
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()
```

## Nested models

Now let's talk about nested models.

Each attribute of pydantic model has a type.

That attribute itself be another Pydantic model.

Using this, we can declare deeply nested JSON objects with specific attribute names, types and validations.

Let's create a model here called Image

```
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
```

Now if you go to the add product route, you will find that fast API would request a body which also accepts the image details.

## Special types.

Apart from values like str, int, float. We can also use more complex singular types that inherit from str.

For example, in the Image model we have declared the url to be a string.

Instead now lets declare it to be of the type HttpUrl.

Here is the code:

```
from pydantic import HttpUrl
class Image(BaseModel):
    url:HttpUrl
    name:str
```

## Attributes with list of submodels.

In the above case, we have used image as a submodel in the Product model.

Instead we can use list of images as a submodel in the Product model.

Here is how.

```
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: List[Image]
```

## Deeply nested models.

Lets say we want to create offers which should contain some of the products from the products model.

Here is how we would do it.

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

Hence now we can say that the models are deeply nested.

Image nested in products, and products nested in offer.