Data Modeling

Introduction to Data Modeling

The first step in every Django project is figuring out the data we want to store.
The database schema.

- What entities /concepts we have
- What fields / attributes
- What relations
- ...
  It really depends on the business rules

Action Plan:

- Find all entities
- Find fields for those entities
- Find relations
- Specify the type of those relations
- Apply normalization to make the database more robust
- Repeat
  Basically the design patterns we followed for SQL classes

🔥 Django is a little cooler

In Django we can define **many-to-many** relations meaning we don't need to manually create three different models to define the **many-to-many** relation. Also the id field is generated automatically in Django. But in scenarios that we need something more complex, we can define it ourselves.

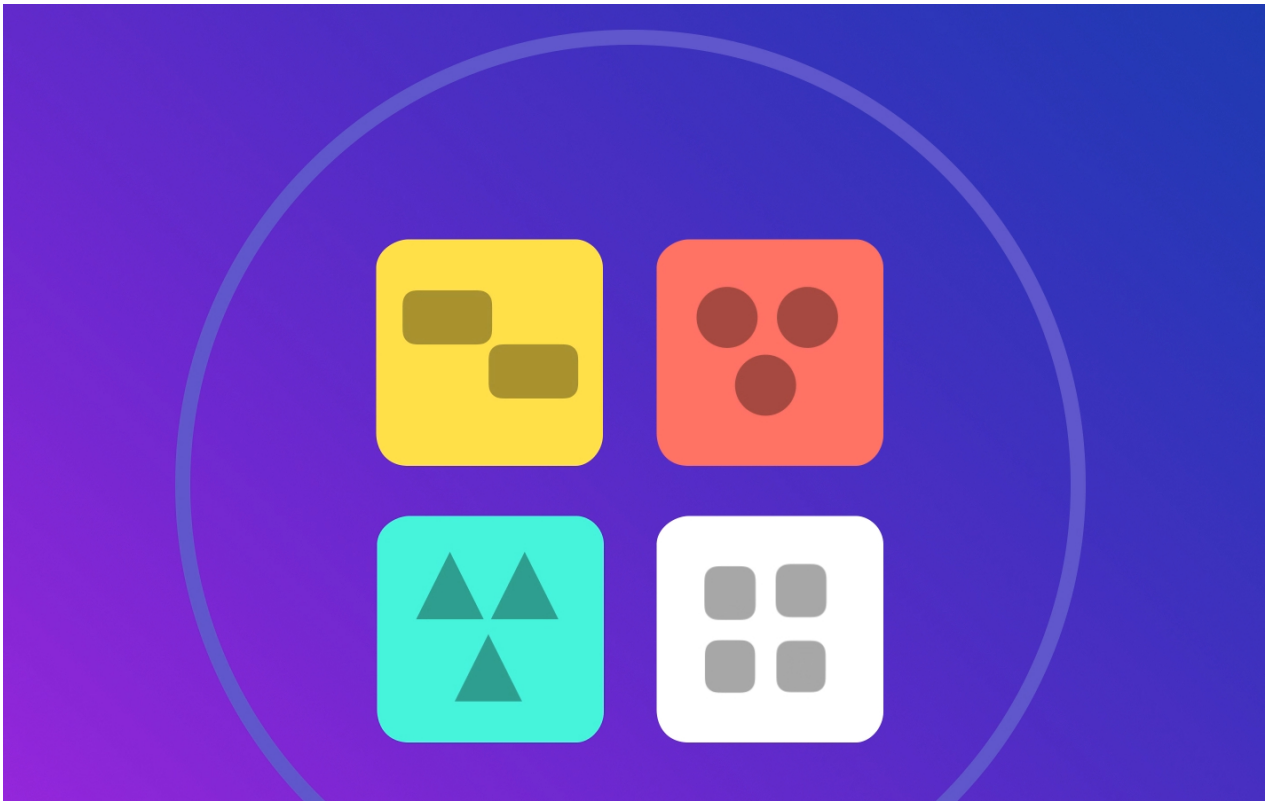
Sometimes Relationships can have attributes. This references the **Product** and **Cart** Relation. where a **product** can be in **Many Carts** and a **Cart** can have **Many Products** in it. but we need to know the quantity of the product. and that's where we define the **CartItem** / **OrderItem** Model. ***CartItem is what we call an Association class***

a **many-to-many** was turned into two **one-to-many** relations

---

Organizing Models in Apps

- Django Projects have multiple apps
- Each apps has it's own functionality
  Which mean, ***Each app has its own data model***



Ways to Organize

- **Monolith** - all models in a single app (including the generic models)
  - Easy to ship
  - Get's more and more complex

    - Hard to maintain
    - Eventually breaks
- **Too many chunks** - too many apps
    - Hard to ship
    - May break on update
    - Coupling
- **Middle-Ground** - Logical Blocks that make sense to be separate
    - Self Contained apps
    - Easy to ship and reuse
    - Zero Coupling
    - i.e: make separate apps for the **Store** and **Tags** related Models, Because tagging is a concept that's abstract and can be used elsewhere (*Blog, Video Streaming, ...*)

MONOLITH ❌
SMALL APPS ❌
MINIMAL COUPLING AND **HIGH COHESION(FOUCS)** ✅

> Unix Rule: Everything should do one thing and do it well.

Creating Models

- Look at Django Docs for creating models with perfect data types and options (Like Indexing a field or default , ...)
- Always use `DecimalField` for monetary values. `FloatField` has rounding issues.

```
class Product(models.Model):
    title = models.CharField(max_length=255)
    description = models.TextField()
    price = models.DecimalField(max_digits=6,
decimal_places=2)
    inventory = models.PositiveIntegerField()
```

```
    last_update = models.DateTimeField(auto_now=True)


class Customer(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    email = models.EmailField(unique=True)
    phone = models.CharField(max_length=11)
    birth_date = models.DateField(null=True, blank=True)
```

Choice Fields

limiting the options of value for a column.

```
YEAR_IN_SCHOOL_CHOICES = {
    "FR": "Freshman",
    "SO": "Sophomore",
    "JR": "Junior",
    "SR": "Senior",
    "GR": "Graduate",
}

YEAR_IN_SCHOOL_CHOICES = [
    ("FR", "Freshman"),
    ("SO", "Sophomore"),
    ("JR", "Junior"),
    ("SR", "Senior"),
    ("GR", "Graduate"),
]
```

Generally, it's best to define choices inside a model class, and to define a suitably-named constant for each value:

```
class Student(models.Model):
    FRESHMAN = "FR"
    SOPHOMORE = "SO"
    JUNIOR = "JR"
    SENIOR = "SR"
    GRADUATE = "GR"
```

```
    YEAR_IN_SCHOOL_CHOICES = {
        FRESHMAN: "Freshman",
        SOPHOMORE: "Sophomore",
        JUNIOR: "Junior",
        SENIOR: "Senior",
        GRADUATE: "Graduate",
    }
    year_in_school = models.CharField(
        max_length=2,
        choices=YEAR_IN_SCHOOL_CHOICES,
        default=FRESHMAN,
    )
```

- When a raw value is **referenced more than once**, it needs to be a **variable** so we don't get in **refactoring hell**.
- The human readable name over here, comes in handy when we start working with the admin interface. we don't want single letters to show up in our lists.

Defining One-to-One Relationships

- Child Model
- Parent Model
  Parent must exist before the child

```
class Address(models.Model):
        customer = models.OneToOneField(
                Customer,
                on_delete=models.CASCADE,
                primary_key=True #! important
        )
```

- Cascade
- Set null
- Set default
- Protect

if we don't use `primary=True`, Django is going to create a ID field and our relationship becomes a one-to-many relationship.

Reverse Relationship

Defining One-to-Many Relationships

```
class Address(models.Model):
        customer = models.ForeignKey(
                Customer,
                on_delete=models.CASCADE
        )
```

If it's not possible for us to define the parent model first, we can pass a string containing the class name in the child.

- Orders are never deleted in e-commerce
- We must have the unit price in 3 places

Defining Many-to-Many Relationships

Promotion -> Product = Many to Many

- a product can have multiple promotions
- a promotion can apply to multiple products

We can define the relationship in either of these models. and Django is going to create the reverse relationship.

- it makes more sense to make it in the products model
- we use plural names because we might have multiple promotions that apply to a single product.
- Reverse relationship in promotion model -> `product_set`

- can be changed with related name in the product model (wherever the relationship is defined)
- Change the default conventions can cause inconsistency so if you want to change it, change it everywhere. *Mosh does it*

```
promotions = models.ManyToManyField(
    Promotion,
    related_name='products'
)
```

Resolving Circular Relationships

Dependencies

Product -> Collection
Collection -> Product = Featured Product

We're gonna face an issue called **Circular Dependency**. Because after all, one of these classes are gonna be defined first which results in the second relation breaking.

> ***A Circular Dependency happens when two models depend on each other at the same time.***

Potential Solution : *Wrap the* `Product` *reference in quotes*
problem with this approach : not updated by refactoring

we have two collection fields now because Django created a reverse relationship with the name `collection`.
Solutions :

- Rename the featured product related name to something else
- Rename the featured product related name to `+` (*Prevents Django from creating the reverse relationship*)

Generic Relationships

Some models can be generic. meaning they're an abstract concepts that can be used at multiple places, Which is a very good thing considering we want shippable apps.
i.e

- Tags
- Likes
- Favorites
- Saved
- ...

For identifying the item being tagged, we need two pieces of information:

- **Object Type** - Table
- **Object ID** - Row
- **Object itself** - for Querying data
  using these we can find any row in any table.

---

for this behavior, instead of using a concrete model like **Product** we use a abstract model called `ContentType`.

This model comes with Django and you can see it in the list of **INSTALLED_APPS** in the **Settings** module. using **content types** we can create generic relationships between our models.

```
from django.contrib.contenttypes.models import ContentType
```

- Content Type - a foreign key with `ContentType`
- ID - Regular integer ID fields (*LIMITATION!*)
- Content - `GenericForeignKey()`

```
class Tag(models.Model):
    label = models.CharField(max_length=255)


class TaggedItem(models.Model):
    tag = models.ForeignKey(Tag, on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType,
on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField() # LIMIT
    object_content = GenericForeignKey()
```

Likes app Concept

```
class LikedItem(models.Model):
    user = models.ForeignKey(User, models.CASCADE)
    content_type = models.ForeignKey(ContentType,
models.CASCADE)
    object_id = models.PositiveIntegerField()
    object_content = GenericForeignKey()
```