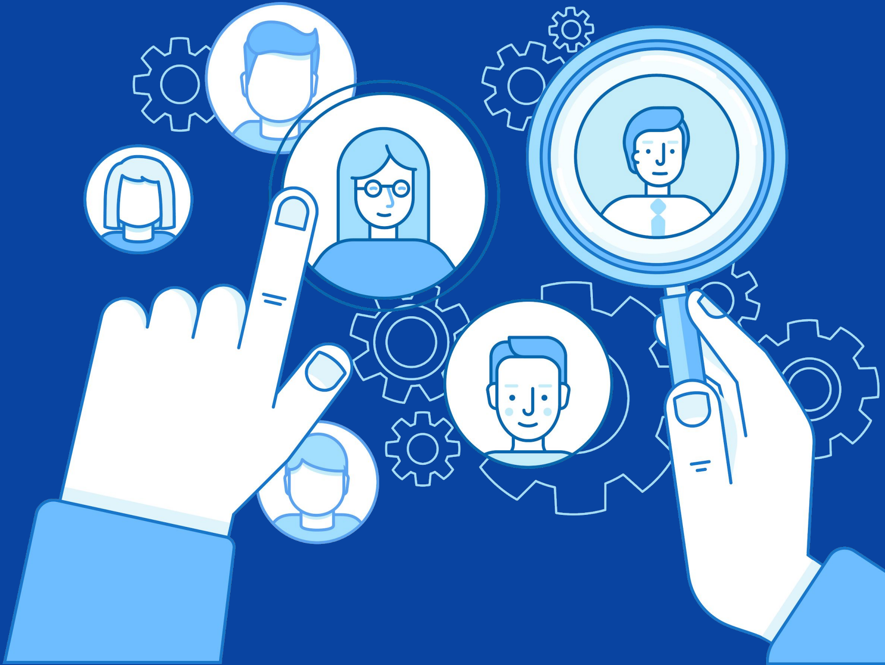
Digital Career Institute

Python Course - Databases - ORM



Digital Career Institute

The Model Meta Class

Digital Career Institute

The Meta Class

The **Meta** class is a class within the Model class.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    ...
    class Meta:
        ...
```

It is used to define some properties of the class (not available on instantiated objects).

Table Names

shop/models.py

```
...

class Item(models.Model):
    """The Item Model."""
    ...
```

```
shop=# \dt
                    List of relations
 Schema |            Name            | Type  |  Owner
------+----------------------------+-------+----------
 blic | auth_group                 | table | postgres
 blic | auth_group_permissions     | table | postgres
 blic | auth_permission            | table | postgres
 blic | auth_user                  | table | postgres
 blic | auth_user_groups           | table | postgres
 public | auth_user_user_permissions | table | postgres
 public | django_admin_log           | table | postgres
 public | django_content_type        | table | postgres
 public | django_migrations          | table | postgres
 public | django_session             | table | postgres
 public | shop_item                  | table | postgres
(11 rows)
```

Django names the table automatically with the pattern `{app_name}_{class_name}` in lowercase.

Table Names

shop/models.py

```
...

class Item(models.Model):
    """The Item Model."""
    ...
    class Meta:
        db_table = "item"
```

The table name can be overridden using the property **`db_table`** of the model **`Meta`** class.

```
hop=# \dt
                   List of relations
hema |             Name             | Type  |  Owner
-----+------------------------------+-------+----------
blic | auth_group                   | table | postgres
blic | auth_group_permissions       | table | postgres
blic | auth_permission              | table | postgres
blic | auth_user                    | table | postgres
public | auth_user_groups           | table | postgres
public | auth_user_user_permissions | table | postgres
public | django_admin_log           | table | postgres
public | django_content_type        | table | postgres
public | django_migrations          | table | postgres
public | django_session             | table | postgres
public | item                       | table | postgres
(11 rows)
```

UI Names

shop/models.py

```
...

class Item(models.Model):
    """The Item Model."""
    ...
    class Meta:
        verbose_name = "Shop item"
        verbose_name_plural = "Shop items"
```

Sometimes, the name given to the class is the best choice at code level but not at UI level. For these occasions, the `verbose_name` and `verbose_name_plural` properties can be used.

Ordering Model Objects

shop/models.py

```
...

class Item(models.Model):
    """The Item Model."""
    ...
    class Meta:
        ordering = ["state"]
```

The default order used when listing objects can be specified using the `ordering` property.

Multiple fields can be used.

Model Inheritance

shop/models.py

```
...

class Item(models.Model):
    """A general item."""
    ...
    class Meta:
        abstract = True

class Tablet(Item):
    """A tablet."""
    inches = models.DecimalField()
    ...
```

Abstract model classes will only exist in the code, usually to provide a set of common fields to various database models that will inherit from them.

Abstract classes will not be created on the database and will not even appear in the migrations.

The extended class will share the same **`Meta`** attributes as the parent class (except the abstract attribute).

Table Indexes

shop/models.py

```
...

class Item(models.Model):
    """A general item."""
    ...
    class Meta:
        indexes = [
            models.Index(fields=["name"]),
            models.Index(fields=["name", "-state"])
        ]
```

Table Indexes

```
models.Index(
     name="<index_name>",
     fields=["<field1>", "<field2>"],
     condition=Q(<field3>="something")
)
```

The **`Index`** class accepts a variety of keyword arguments to adapt the index creation to the needs of the model.

For instance, to create partial indexes.

PostgreSQL Table Indexes

shop/models.py

```
from django.contrib.postgres.indexes import GistIndex

class Item(models.Model):
    """A general item."""
    ...
    class Meta:
        indexes = [
            GistIndex(fields=["location"])
        ]
```

Django also provides classes to create PostgreSQL specific index methods.

Constraints

shop/models.py

```
from django.db.models import CheckContraint, Q

class NewItem(models.Model):
    """A new item."""
    ...
    class Meta:
        constraints = [
            CheckConstraint(check=Q(state="New"),
                            name="new_items_only")
        ]
```

Multi-Column Unique Constraints

shop/models.py

```
from django.db.models import UniqueContraint, Q

class NewItem(models.Model):
    """A new item."""
    ...
    class Meta:
        constraints = [
            UniqueConstraint(fields=["name", "state"],
                                    name="unique_state_name")
        ]
```

Customizing Models

Object Custom Save

The **`save`** method of the model objects may be overridden to include additional instructions before or after the actual save.

It can also be overridden to condition the saving to some additional criteria.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    ...

    def save(self, *args, **kwargs):
        # instructions before save
        self.last_updated_by = self.request.user
        # save the object
        super().save(*args, **kwargs)
        # instructions after save
```

Object String Representation

```
>>> tablet
<Item: Item object (1)>
>>>
```

```
>>> tablet
<Item: New Tablet>
>>>
```

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    name = models.CharField(
        max_length=100)
    state = models.CharField(
        max_length=10)

    def __str__(self):
        return f"{self.state} {self.name}"
```

The `__str__` Python method can also be used in models to provide a more readable reference to the actual object.

Model + Form = ModelForm

Models often require forms to be able to enter and update data without typing code.

Django provides a specific class for these cases that makes it much easier to work with forms and models.

Django will automatically use the default form field type for each model field in the **`meta.fields`** attribute.

Additional fields can be added as attributes of the **`ModelForm`**.

shop/forms.py

```
from django.forms import ModelForm
from shop.models import Item


class ItemForm(ModelForm):
    class Meta:
        model = Item
        fields = ["name", "state"]
```

Model Validation

Model validation is only automatically done when saving the associated `ModelForm`, not when saving the `Model` directly.

```
>>> Item.objects.create(name="Phone", state="New")
Saving Model
<Item: New Phone>
>>> ItemForm({"name":"Phone", "state":"New"}).save()
Saving ModelForm
Validating Model
Saving Model
>>>
```

Model Validation

The model can automatically be validated on model save, by overriding the **`save`** method and calling the **`full_clean`** method before saving.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    ...

    def save(self, *args, **kwargs):
        self.full_clean()
        super().save(*args, **kwargs)
```

Model Validation

Individual field validation can be done the same way as with forms, using validators.

shop/models.py

```
def only_newish(value):
    if value not in ["New", "Like new"]:
        Raise ValidationError(...)

class Item(models.Model):
    """The Item Model."""
    name = models.CharField(max_length=100)
    state = models.CharField(
        max_length=10,
        validators=[only_newish]
    )
```

Model Validation

A general validation can be done by using the `clean` method, usually when it involves conditions on more than one field.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    name = models.CharField(max_length=100)
    state = models.CharField(max_length=10)

    def clean(self, *args, **kwargs):
        if … :
            Raise ValidationError(…)
```

Model Custom Methods

If there are recurring operations done with the same objects, these can be implemented as model methods.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""

    def my_custom_method(self, *args, **kwargs):
        …
```

```
>>> item = Item.objects.get(pk=1)
>>> result = item.my_custom_method()
```

Custom Model Manager

Overriding the **`get_queryset`** method will override all querysets returned by the **`objects`** model manager.

```
>>> Item.objects.all().values()
<QuerySet [{'id': 1, 'name':
'Tablet', 'state': 'New',
'full_name': 'New Tablet'}, ...
```

Specific manager methods (**`get`**, **`all`**, **`filter`**,...) can also be overridden.

shop/models.py

```
class CustomManager(models.Manager):
    def get_queryset(self):
        # Generate a virtual full_name field
        qs = super().get_queryset()
        return qs.annotate(
            full_name=models.functions.Concat(
                "state", models.Value(" "),
                "name")
        )

class Item(models.Model):
    """The Item Model."""
    objects = CustomManager()
```

Custom Model Manager

The default property **`objects`** can be renamed.

If a model manager is assigned to a property different than **`objects`**, this one will not be set and will raise an error if it is being accessed.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    objekte = models.Manager()
```

```
>>> Item.objekte.all().values()
<QuerySet [{'id': 1, 'name': 'Tablet', 'state':
'New'}, {'id': 2, 'name': 'Tablet', 'state':
'Old'}, ...
>>> Item.objects.all().values()
AttributeError: type object 'Item' has no
attribute 'objects'
```

Custom Model Manager

A model may have more than one manager.

shop/models.py

```
class PhoneManager(models.Manager):
    def get_queryset(self):
        # Only phones
        qs = super().get_queryset()
        return qs.filter(name="Phone")


class Item(models.Model):
    """The Item Model."""
    objects = models.Manager()
    phones = PhoneManager()
```

```
>>> Item.phones.all().values()
<QuerySet [{'id': 4, 'name':
'Phone', 'state': 'New'}, ...
```

We learned ...

- What is an Object-Relational Mapping.
- That the ORM integrates into the Models in the MVC design pattern.
- How to define models and use migrations to synchronize them with the database.
- How to use the ORM to manage and query data.
- That QuerySets can be reused many times.
- That lookups allow us to define special operators and can span to the related tables.
- That models can be highly customized.

Model Fields

Model Fields

Model fields have a direct relationship with a column in a database table.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    name = models.CharField(max_length=100)
    state = models.CharField(max_length=50)
```

```
CREATE TABLE shop_item (
  id    serial NOT NULL PRIMARY KEY,
  name  varchar(100) NOT NULL,
  state varchar(50) NOT NULL
);
```

Model Fields

Model fields have a direct relationship with a column in a database table.

shop/models.py

```
class Item(models.Model):
    """The Item Model"""
    name = models.Ch
    state = models.C
```

The ORM automatically creates a field named **`id`** as primary key.

```
CREATE TABLE shop_item (
  id    serial NOT NULL PRIMARY KEY,
  name  varchar(100) NOT NULL,
  state varchar(50) NOT NULL
);
```

Model Fields

Model fields have a direct relationship with a column in a database table.

shop/models.py

```
class Item(models.Model):
    """The Item Model"""
    name = models.Ch
    state = models.C
```

By default, all fields are defined as **`NOT NULL`**.

```
CREATE TABLE shop_item (
  id    serial NOT NULL PRIMARY KEY,
  name  varchar(100  NOT NULL,
  state varchar(50) NOT NULL
);
```

Model Fields

Model fields have a direct relationship with a column in a database table.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    name = models.CharField(max_length=100)
    state = models.CharField(max_length=50)
```

```
CREATE TABLE shop_item (
  id     serial
  name   varcha
  state  varcha
);
```

The **`CharField`** type requires an argument **`max_length`**.

Common Field Arguments

All field types share some common optional **keyword arguments**.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    name = models.CharField(
        max_length=100,
        null=True,                          # default False
        blank=True,                         # default False
        choices=SIZES,                      # default None
        default="Tablet",                   # default None
        help_text="The item name",          # default None
        primary_key=True,                   # default False
        unique=True                         # default False
        …
    )
```

The Null & Blank Arguments

If **`True`**, the **`null`** argument allows storing null values on the database.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    name = models.CharField(max_length=100)
    state = models.CharField(max_length=2,  null=True,
                                 blank=True)
```

If **`True`**, the **`blank`** argument allows submitting the form with empty values.

Setting **`null=False`** and **`blank=True`** on a **`varchar`** field will let the user leave the form field empty and an empty string will be stored instead of **`NULL`**.

The Choices Argument

Choices can be defined using a tuple of tuples.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    STATES = (
        ("N", "New"),
        ("LN", "Like new"),
        ("R", "Refurbished"),
        ("U", "Used"),
    )
    name = models.CharField(max_length=100)
    state = models.CharField(max_length=2,  choices=STATES)
```

Values

Labels

Primary Keys, Unique & Indexes

The **`primary key`** argument sets a custom primary key.
The **`unique`** argument sets a unique constraint.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    code = models.CharField(max_length=10,  primary_key=True)
    name = models.CharField(max_length=100,  unique=True)
    state = models.CharField(max_length=20,  db_index=True)
```

The **`db_index`** argument will create an index on the field.

Field Types

Django’s ORM offers a variety of field types.

- AutoField
- BigAutoField
- BigIntegerField
- BinaryField
- BooleanField
- CharField
- DateField
- DateTimeField
- DecimalField
- DurationField
- EmailField
- FileField and FieldFile
- FilePathField
- FloatField
- ImageField
- IntegerField
- GenericIPAddressField
- JSONField
- NullBooleanField
- PositiveBigIntegerField
- PositiveIntegerField
- PositiveSmallIntegerField
- SlugField
- SmallAutoField
- SmallIntegerField
- TextField
- TimeField
- URLField
- UUIDField

Text Fields

The **`CharField`** type is equivalent to the **`varchar`** database type.
The **`TextField`** is equivalent to the **`text`** database type.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    name = models.CharField(max_length=100)
    description = models.TextField()
```

The **`CharField`** will render as a **`TextInput`** whereas the **`TextField`** will render as a **`Textarea`**.

The **`CharField`** has a required argument **`max_length`** and the **`TextField`** has no required argument.

Integer Fields

The **`IntegerField`**, **`BigIntegerField`** and **`SmallIntegerField`** types allow the creation of integer fields of different sizes.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    amount = models.SmallIntegerField()  # smallint
    total_sales = models.IntegerField()  # int
    total_revenue = models.BigIntegerField()  # bigint
```

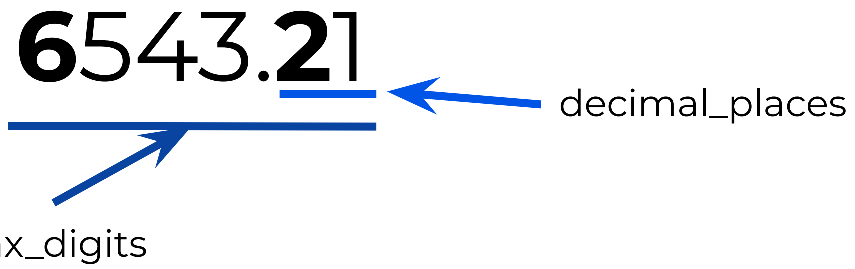
A subclass for each of these fields is available for integers above or equal to 0, as **`PositiveIntegerField`**, **`PositiveBigIntegerField`** and **`PositiveSmallIntegerField`**.

Real Numbers Fields

The **`DecimalField`** type is equivalent to Python's **`decimal`** type.
It requires the arguments **`max_digits`** and **`decimal_places`**.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    price = models.DecimalField(
        max_digits=6, decimal_places=2
    )
```



Real Numbers Fields

The **`FloatField`** type is equivalent to Python's **`float`** type.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    price = models.FloatField()
```

Time Related Fields

Django's ORM offers a variety of time related types:
`DateField`, **`DateTimeField`**, **`TimeField`** and **`DurationField`**.

shop/models.py

```
class Employee(models.Model):
    """The Employee Model."""
    date_of_birth = models.DateField()
    enroled_on = models.DateTimeField()
    daily_work_start = models.TimeField()
    hourly_work_balance = models.DurationField()
```

Date and time types accept the optional arguments:

- **`auto_now`**: If **`True`**, automatically set the value to ***now*** every time the model is saved.
- **`auto_now_add`**: If **`True`**, automatically set the value to ***now*** when the record is first created.

Boolean Fields

Boolean types are implemented using the **`BooleanField`** type.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    is_on_stock = models.BooleanField()
    order_in_process = models.BooleanField(null=True)
```

Boolean fields will be rendered as checkboxes if the field does not allow **`NULL`**s. If it does, they will be rendered as dropdowns with three options: *Unknown*, *Yes* and *No*.

File Related Fields

Django's ORM uses **FileField** to upload and store files.
A special type **ImageField** is available for image files.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    brochure = models.FileField()
    picture = models.ImageField(upload_to="img")
```

These types do not support the **primary_key** argument.

The **upload_to** argument can be used to specify a subdirectory inside the directory declared in the **settings.MEDIA_ROOT** path.

Media Files: Development

Files uploaded by the user are called **Media Files**.
In development, they are served by Django and defined in the settings.

project/settings.py

```
…
MEDIA_URL = "/media/"
MEDIA_ROOT = os.path.join(BASE_DIR, "media")
```

project/urls.py

```
from django.conf.urls.static import static
# code before
urlpatterns = patterns('',
    # View paths
) + static(settings.MEDIA_URL,
           document_root=settings.MEDIA_ROOT)
```

Media Files: Production

In production, media files are served, like static files, by the web server.

project/settings.py

```
…
MEDIA_URL = "/media/"
MEDIA
```

proje

```
from
# code before
urlpatterns = patterns('',
    # View paths
) + static(settings.MEDIA_URL,
           document_root=settings.MEDIA_ROOT)
```

The media files settings
will **not** be used in production.

File Related Fields

Accessing fields of the types **`FileField`** and **`ImageField`** will return a **`FieldFile`** object.

```
>>> item = Item.objects.get(pk=1)
>>> field_file = item.brochure
>>> print(field_file.name)
item1.pdf
>>> print(field_file.path)
/media/item1.pdf
```

The **`FieldFile`** object offers methods and properties to deal with files (save, delete, open, close, url, name, path,...).

File Related Fields

Django's ORM offers an additional type named **`FilePathField`**.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    brochure = models.FileField()
    picture = models.ImageField(upload_to="img")
    warranty = models.FilePathField(path="/warranty")
```

The **`FilePathField`** type will render as a select whose options will be the files present in the **required** **`path`** argument.

Additional Types: Auto Fields

The **`AutoField`** and **`BigAutoField`** are subclasses of the **`IntegerField`** and **`BigIntegerField`**.

shop/models.py

```
class Item(models.Model):
    """The Item Model."""
    my_custom_id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=100)
```

This field type is rarely used, as defining a model without primary key will automatically create a primary key named **`id`** of type **`BigAutoField`**.

Additional Types: Text Related Fields

The **`EmailField`**, **`URLField`** and **`SlugField`** are stored as text fields.

shop/models.py

```
class Profile(models.Model):
    """The User Profile Model."""
    email = models.EmailField(unique=True)
    website = models.URLField()
    public_profile = models.SlugField()
```

They provide specific validation for these type of strings.

Additional Types: JSON Fields

The `JSONField` allows storing JSON strings.

shop/models.py

```
class Profile(models.Model):
    """The User Profile Model."""
    data = models.JSONField()
```

On PostgreSQL, it uses the database type `jsonb`.

Additional Types: JSON Fields

The contents of the **`JSONField`** can be queried using lookups.

```
>>> Item.objects.create(data={"key1": "value1"})
>>> value1 = Item.objects.filter(data__key1="value1")
>>> key1 =  Item.objects.filter(data__has_key="key1")
```

This field type also has specific lookups, such as **`has_key`**, that will return all records whose JSON field has a specific key.

We learned ...

- That Django’s ORM provides a variety of field types for every need.
- That there’s a variety of integer and decimal field types that can be used to optimize the database.
- That dropdown widgets can be defined using the `choices` option of a field type.
- How to define primary keys, unique fields and indexes.
- How to define file input fields and the path to store them.

Digital Career Institute

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute