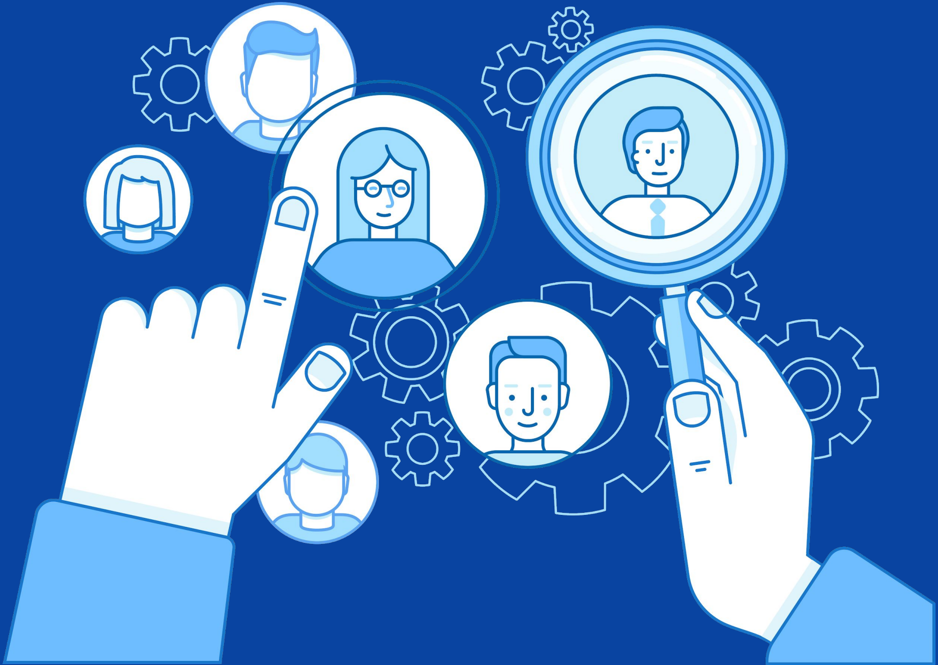
Digital Career Institute

Python Course - Databases - ORM



Digital Career Institute

Goal of the Submodule

The goal of this submodule is to introduce the learners to working with databases using the Object-Relational Mapping in Django. By the end of this submodule, the learners will be able to:

- Define models in Django
- Synchronize the models with the database.
- Use the ORM to manage and query the data.
- Use and customize the Administration Site.

Digital Career Institute

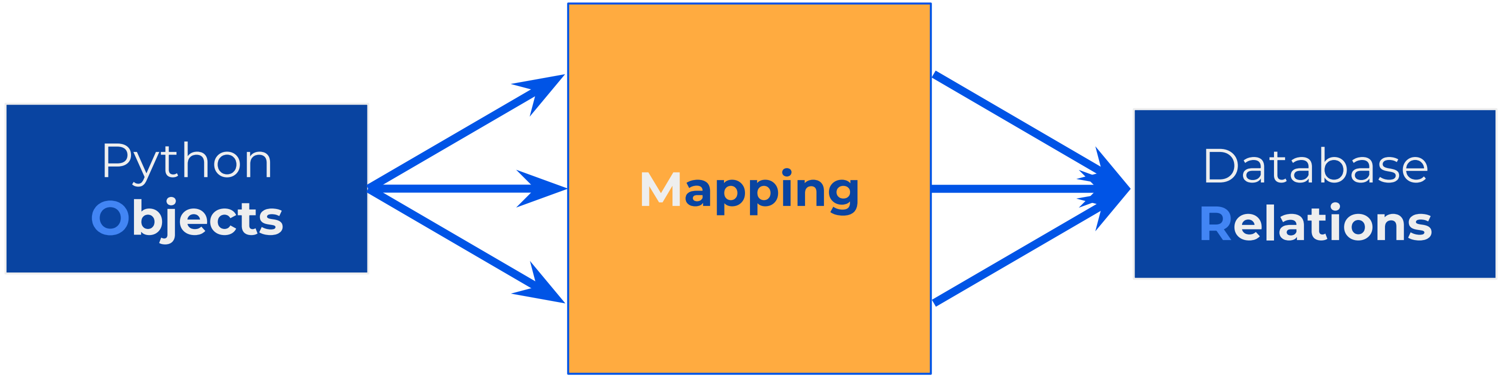
Topics

- The Object-Relational Mapping (ORM).
- Django models and migrations.
- Model Manager and QuerySets.
- The Meta Class.
- Model customization.
- Model field types.
- One-to-many model relationships.
- Forward and backwards relation.
- One-to-one model relationships.
- QuerySet evaluation.
- Django administration site.

Introduction to ORM

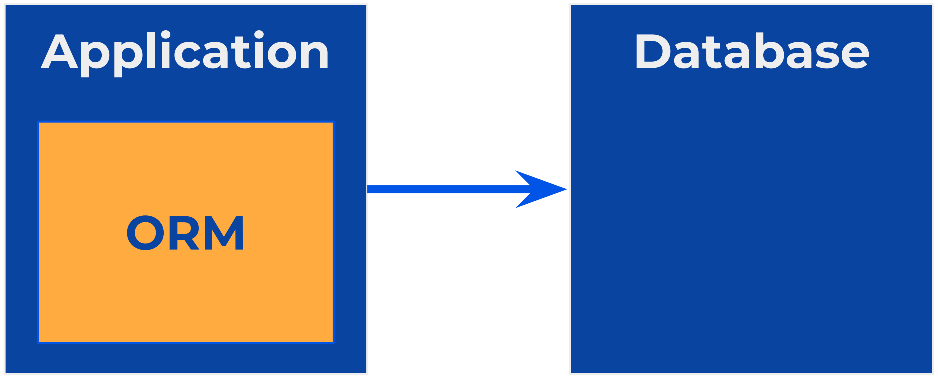
O*bject-****R****elational* ***M****apping*

Object-Relational Mapping



An **ORM** is a software written in a particular programming language that provides an OOP interface to manipulate and query the database relations (tables).

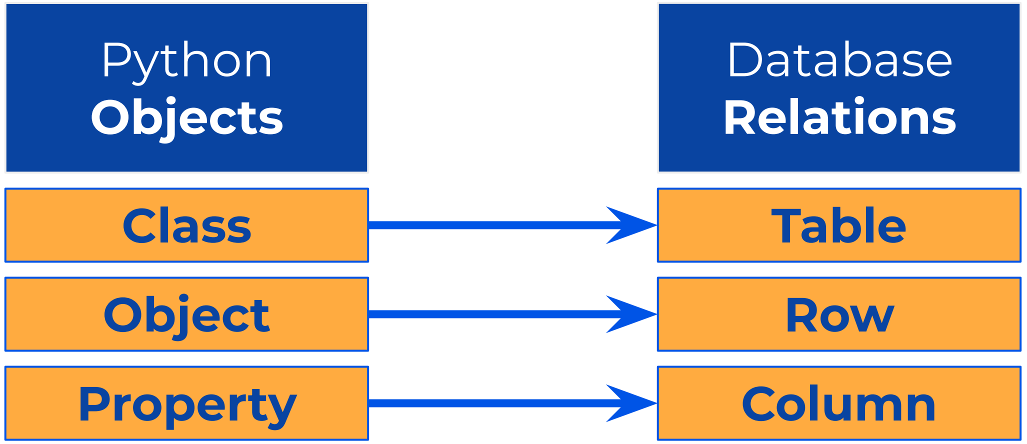
Object-Relational Mapping



The **ORM** is part of the application codebase.

It is used to simplify database operations in the application.

Object-Relational Mapping



An **ORM** defines classes for each table
and properties for each column.
Instantiating a new object is like adding a row to the table.

Object-Relational Mapping

Python
Objects

```
import orm
class People(orm.Table):
    id = orm.IntegerField()
    first = orm.CharField(20)
    last = orm.CharField(50)
```

Database
Relations

```
CREATE TABLE people (
  id     integer,
  first  varchar(20),
  last   varchar(50)
);
```

The **ORM** can produce any required operation on the database once a class has been defined.

Object-Relational Mapping

Python **Objects**

```
mary = People(id=1,
              first="Mary",
              last="Hleb")
```

Database **Relations**

```
INSERT INTO people
VALUES (
  1, "Mary", "Hleb"
);
```

The **ORM** simplifies how we store and use data within the application. In a single instruction the data is inserted and the corresponding Python object is returned.

Advantages of an ORM

1. It provides the opportunity to use a single language both for the application and the database operations, that will simplify development.

2. It often provides a cleaner interface to complex queries, thus often producing a more readable code that will be easier to maintain and troubleshoot.

3. It makes it easier to reuse queries. Some common queries can be stored as variables and reused with a very simple and readable code.

4. Altogether, it helps increase the developer's productivity by simplifying the most common database operations in an application.

Disadvantages of an ORM

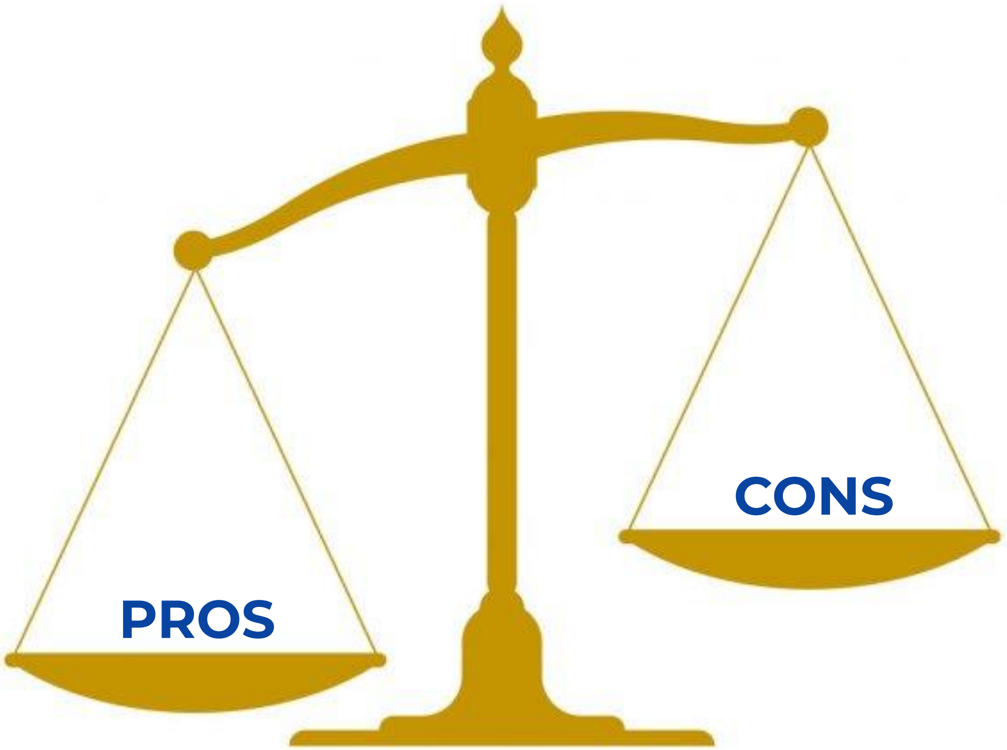
1. Sometimes (not always), it may require some additional work to define the classes in the code, but Django's ORM has reduced this to a minimum by providing useful tools that sometimes make the process much faster.

2. Some of the most uncommon SQL operations may still need to be done using SQL.

3. Because it makes the code look simpler, the actual SQL being executed may go unnoticed and special attention should be paid to each operation done with the ORM and its SQL equivalent.

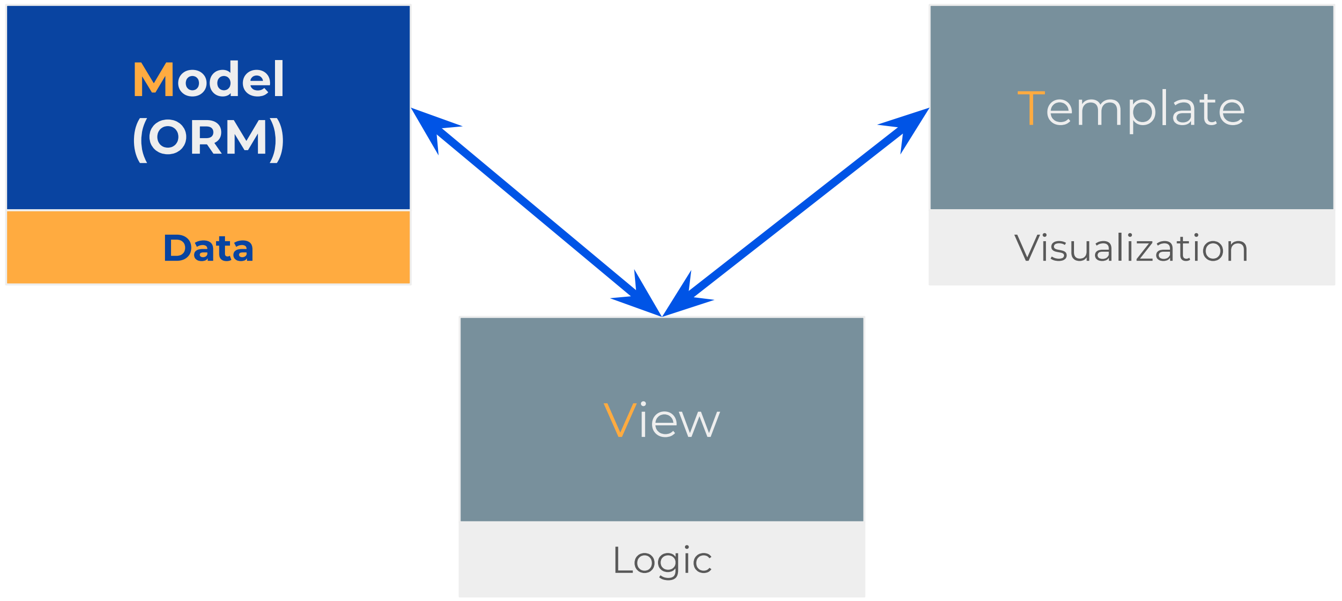
Advantages vs. Disadvantages

DLI



Django’s ORM

The **ORM**, in a *Model-View-Controller* framework like Django, integrates into the **Models**.



Introduction to Django Models

Database Setup: Installed Apps

project/settings.py

```
INSTALLED_APPS = [
    'shop',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

To use models the app must be registered in the `settings.py` file.

PostgreSQL Setup

project/settings.py

```
DATABASES = {
     'default': {
          'ENGINE': 'django.db.backends.postgresql_psycopg2',
          'NAME': '<db_name>',
          'USER': '<db_username>',
          'PASSWORD': '<password>',
          'HOST': '<db_hostname_or_ip>',
          'PORT': '<db_port>',
     }
}
```

To be able to use models with PostgreSQL, the settings must be defined using Django's database backend **`postgresql_psycopg2`**.

PostgreSQL Setup

For the previous database settings to work, the Python package **psycopg2-binary** should be installed in the environment.

```
(python3env)$ pip install psycopg2-binary
```

Ideally, it should be added to the **requirements.txt** file.

```
(python3env)$ pip freeze | grep psycopg >> requirements.txt
```

Django Models

```
+ shop
  + migrations
    - __init__.py
  - __init__.py
  - admin.py
  - apps.py
  - models.py
  - tests.py
  - views.py
```

In Django, models are usually defined in the file **`models.py`** on each app directory.

Every app usually manages its own models.
This helps keeping the code organized.

Django Models

shop/models.py

```
from django.db import models


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

Django's `models` is a module that can be imported from `django.db`.

Each model is a named class that instantiates from the `models.Model` superclass. Each class name will match a database table of the *same* name.

Properties in the class can be defined as columns in the table using a variety of field types.

Model ←-→ Database Synchronization

Digital Career Institute

Model → Database: Migrations

Django comes with tools to synchronize models and database.
To bring the model changes to the database, **migrations** are used.

Migrations are a record of the changes in the models.

shop/migrations/0001_initial.py

```
class Migration(migrations.Migration):
    initial = True
    dependencies = []
    operations = [
        migrations.CreateModel(
            name='Item',
            fields=[...],
        ),
    ]
```

Creating Migrations

Migrations can be created with the **`makemigrations`** command.

```
$ python3 manage.py makemigrations
Migrations for 'shop':
  shop/migrations/0001_initial.py
    - Create model Item
```

The code can be manually typed, but very often it is easier to execute the **`makemigrations`** command to produce the migration code.

Migrations Naming

The **`makemigrations`** command automatically generates a name for the migration file.

```
$ python3 manage.py makemigrations
Migrations for 'shop':
  shop/migrations/0001_initial.py
    - Create model Item
```

An identifier, internal to the app.

A string with the main changes.

Running Migrations

Migrations then need to be applied to the database. This is done with the **`migrate`** command.

```
$ python3 manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, ...
Running migrations:
  Applying shop.0001_initial... OK
```

Creating Additional Migrations

The **makemigrations** command will detect any changes in the models and will record them.

shop/models.py

```
from django.db impo

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

```
$ python3 manage.py makemigrations
Migrations for 'shop':
  shop/migrations/0002_remove_item_stock.py
    - Remove field stock from item
```

The stock field has been removed.

Migration Dependencies

shop/migrations/0002_remove_item_stock.py

```
class Migration(migrations.Migration):
    dependencies = [
        ('shop', '0001_initial'),
    ]
    operations = [
        migrations.RemoveField(
            model_name='item',
            name='stock',
        ),
    ]
```

Django will only run a migration if its dependencies have already been ran.

Merging Migrations

```
$ python3 manage.py squashmigrations shop 0002
Will squash the following migrations:
 - 0001_initial
 - 0002_remove_item_stock
Do you wish to proceed? [yN] y
Optimizing…
  Optimized from 2 operations to 1 operations.
Created new squashed migration 0001_squashed_0002_remove_item_stock.py
  You should commit this migration but leave the old ones in place;
  the new migration will be used for new installs. Once you are sure
  all instances of the codebase have applied the migrations you squashed,
  you can delete them.
```

The parameters of the command are the name of the app (`shop`), the id of the initial migration (optional) and the id of the last one.

Merging Migrations

shop/migrations/0001_squashed_0002_...

```
class Migration(migrations.Migration):
    replaces = [
        ('shop', '0001_initial'),
        ('shop', '0002_remove_item_stock')
    ]
    initial = True

    ...
```

Removing all the files in the migrations directory and running `makemigrations` again will also work if we want to merge everything, but if there are multiple developers this may pose a risk of migration inconsistency.

Migrations: Why 2 Commands?

Generating and exposing the migration files helps keep track of the changes done on the database and models, and gives more control to the developer.

In some stages of the development, little modifications may be done often in the models, requiring every time a database synchronization to test the changes and generating many irrelevant files that just add noise to the codebase.

To keep the code tidier and if the database has no data or the loading process is streamlined, it may seem preferable to simply edit the main migration file and rebuild the database entirely.

A streamlined migration process can always be done using a simple bash script or an alias.

Database → Model: Inspect DB

To convert the database tables into models, **`inspectdb`** is used.

```
(env)$ python3 manage.py inspectdb > models.py
```

When working with *legacy databases* (they existed before the app), **`inspectdb`** will speed up the initial model setup.

The resulting models will often be reviewed to make sure everything is as desired.

Data Migration: Fixtures

To export the database data, **`dumpdata`** is used.

```
(env)$ python3 manage.py dumpdata > data.json
```

To import the data into a new instance, **`loaddata`** is used.

```
(env)$ python3 manage.py loaddata data.json
Installed 9 object(s) from 1 fixture(s)
```

The Model Manager & QuerySets

Digital Career Institute

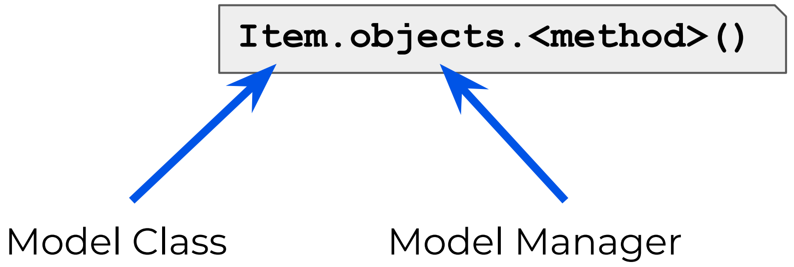
The Model Manager

The model **Manager** is a property of the **`Model`** class.

The model manager is, on its turn, a class with **methods to create, delete and access** rows in the database table.

The Model Manager

The property holding the default manager is named **`objects`**.



```
Item.objects.<method>()
```

The Model Manager: Methods

The most basic methods of the model manager are:

CREATE

GET

FILTER

EXCLUDE

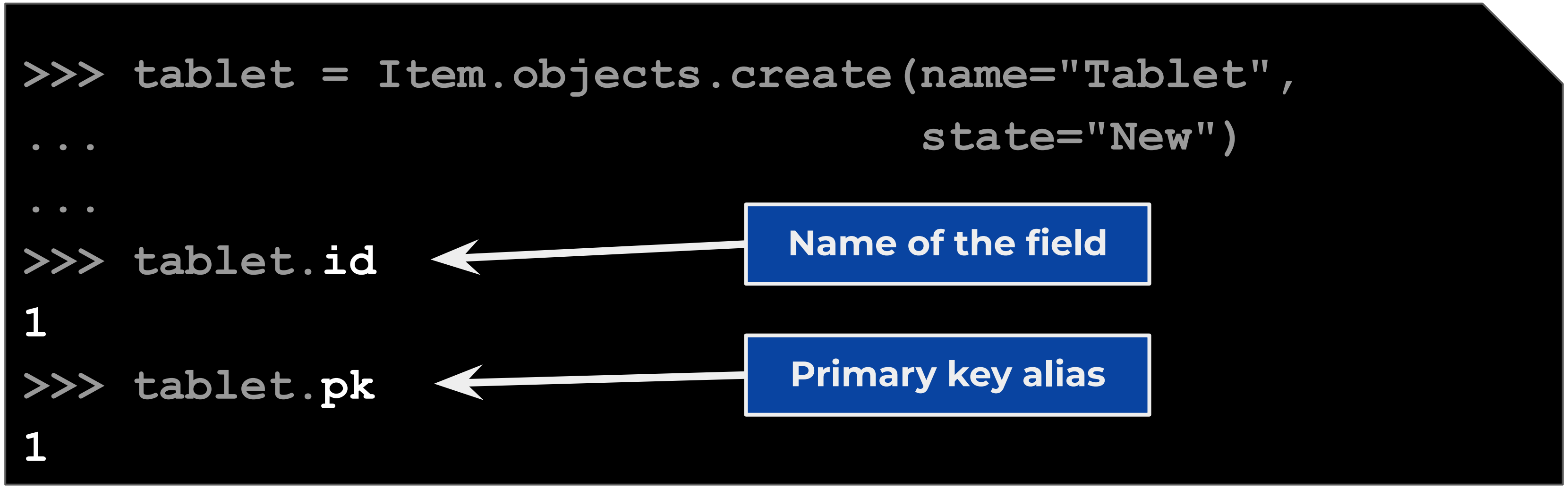
ALL

Create

```
(env)$ python3 manage.py shell
>>> from shop.models import Item
>>> tablet = Item.objects.create(name="Tablet",
...                                    state="New")
...
>>> tablet
<Item: Item object (1)>
>>>
```

The model manager has a method named `create`. This method will store the new data in the database and will return the model object.

Create



```
>>> tablet = Item.objects.create(name="Tablet",
...                                state="New")
...
>>> tablet.id
1
>>> tablet.pk
1
```

A table created with the ORM will always have a primary key column. If one has not been defined in our code, the ORM will automatically add one named `id` (with an alias named `pk`).

If the model specifies a primary key with a name different than `id`, `pk` will point to that field.

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute