Connecting Django to MySQL

Install:

```
pip install mysqlclient
```

Configure in `settings.py`

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'db_name',
        'USER': 'root',
        'PASSWORD': 'yourpass',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

---

1) What are Models?

In Django, **models are Python classes that define the structure of your database tables**.

Example:

1) `models.py` (example model)

python Copy code

```python
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=10, decimal_places=2)  # better for money
    description = models.TextField(blank=True)                    # optional
    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

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

Notes:

- `CharField` for short text, `DecimalField` for money (avoid `FloatField` for currency).
- `blank=True` lets the field be empty in forms; `null=True` would allow NULL in DB (use carefully).
- `auto_now_add` sets creation time once; `auto_now` updates timestamp on every save.
- `__str__` helps readable names in admin and shell.

- Each **class = one table**
- Each **attribute = one column**
- Django automatically creates tables based on these models (`migrate`)

---

2) Why not Raw SQL?

You *can* use raw SQL, but it has problems:

Problem with raw SQL	Django Model Advantage
Hard to read/write	Models are Pythonic and readable

Database specific syntax	Models work with any DB (MySQL/SQLite/Postgres)
No safety checks	Django handles escaping & prevents SQL injection
Hard to maintain	Easy to update using migrations
Manual table creation	Auto table creation via `migrate`

---

3) Common Django Model Fields

Field	Purpose
`CharField`	Short text (name, title)
`TextField`	Large text (description)
`IntegerField`	Numbers
`FloatField`	Decimal values
`BooleanField`	True/False
`DateField`, `DateTimeField`	Dates and timestamps
`FileField`, `ImageField`	Files & Images
`ForeignKey`	One-to-many relation
`ManyToManyField`	Many-to-many relation
`OneToOneField`	One-to-one relation

---

4) What is ORM?

ORM (Object Relational Mapping) means we interact with the database using Python objects instead of SQL.

Example (without SQL):

```
Product.objects.create(name="Pen", price=10)
```

This creates a new row in DB without writing:

```
INSERT INTO product (name, price) VALUES ("Pen", 10);
```

ORM converts Python → SQL internally.

---

5) Important ORM Methods

Method	Use
`create()`	Insert new record
`all()`	Get all rows
`filter()`	Conditional query (returns multiple)
`get()`	Fetch single row (error if not found)
`update()`	Update multiple records
`delete()`	Delete records
`order_by()`	Sort results
`exists()`	Check if records exist

Migration Process (How it works)

1. **Create / change model** in `models.py`
2. Run `python manage.py makemigrations`

→ Django generates migration files (Python code of SQL changes)

3. Run `python manage.py migrate`

→ Executes that migration and creates/updates tables in MySQL

Importing Model in Views

```
from .models import Product
```

Then use it:

```
Product.objects.create(name="Book", price=120)
products = Product.objects.all()
```

Crud Operations:

✅ urls.py — (Separate URLs for each operation)

```
from django.urls import path
from . import views

urlpatterns = [
    path('products/', views.list_products, name='product-list'),
    path('products/create/', views.create_product,
name='product-create'),
    path('products/<int:id>/', views.get_product,
name='product-detail'),
    path('products/<int:id>/update/', views.update_product,
name='product-update'),
    path('products/<int:id>/delete/', views.delete_product,
name='product-delete'),
]
```

✅ views.py — CRUD with `request.body` JSON (NO serializers)

```
import json
from django.http import JsonResponse, HttpResponseNotAllowed
from django.shortcuts import get_object_or_404
from .models import Product


def list_products(request):
    if request.method != "GET":
        return HttpResponseNotAllowed(['GET'])
    data = list(Product.objects.values())  # list of dicts
    return JsonResponse(data, safe=False)


def get_product(request, id):
    if request.method != "GET":
        return HttpResponseNotAllowed(['GET'])
    product = get_object_or_404(Product, id=id)
    return JsonResponse({
        "id": product.id,
        "name": product.name,
        "price": product.price,
    })


def create_product(request):
    if request.method != "POST":
        return HttpResponseNotAllowed(['POST'])
    body = json.loads(request.body)
    p = Product.objects.create(
        name=body["name"],
        price=body["price"]
    )
    return JsonResponse({"message": "created", "id": p.id})
```

```
def update_product(request, id):
    if request.method != "PUT":
        return HttpResponseNotAllowed(['PUT'])
    body = json.loads(request.body)
    product = get_object_or_404(Product, id=id)

    product.name = body.get("name", product.name)
    product.price = body.get("price", product.price)
    product.save()

    return JsonResponse({"message": "updated"})


def delete_product(request, id):
    if request.method != "DELETE":
        return HttpResponseNotAllowed(['DELETE'])
    product = get_object_or_404(Product, id=id)
    product.delete()
    return JsonResponse({"message": "deleted"})
```

✅ Quick Explanation: “Style-2” way

Instead of having:

```
/products/5/update/
/products/5/delete/
```

We could use a **single URL and use HTTP method to decide**:

```
/products/5/    → GET = detail, PUT = update, DELETE = remove
```

This is the **RESTful convention**.
Code logic is identical — only URLs change and branching happens in **one view** using `if request.method == ...`.