**Njango ORM System Documentation**

**This document explains the structure, features, and usage of your Django-inspired ORM system for Node.js, built on top of TypeORM. It allows you to define models and query data using a familiar, Django-like API.**

1. **Overview**

* **Model Definition: Define models using a static**[**fields**](vscode-file://vscode-app/c:/Users/MAZHAR/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)**property, similar to Django.**
* **Field Types: Use a rich set of field types and relationships.**
* **Query API: Access data using an**[**objects**](vscode-file://vscode-app/c:/Users/MAZHAR/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)**manager with methods for querying, creating, updating, deleting, and aggregating.**
* **Lookups: Use Django-style lookups (\_\_contains, \_\_lt, etc.) for advanced filtering.**
* **Q Object: Build complex queries with logical operators.**

**1️⃣ Fields (Columns in your database)**

Fields are the **data you store in a table**. Think of a table like an **Excel sheet**, and each column is a field.

**Examples:**

javascript

CopyEdit

class Product extends Model {

static fields = {

name: fields.CharField({ length: 100 }), // Text field for product name

description: fields.TextField(), // Long text

price: fields.FloatField(), // Price of product

in\_stock: fields.BooleanField({ default: true }), // Is the product available?

created\_at: fields.DateTimeField(), // Date and time created

}

}

**Use Case:**  
You can store products in your store with name, description, price, stock status, and creation date.

**2️⃣ Relations (Connecting tables)**

Relations are like **links between tables**, similar to Excel's “lookup” or “reference”.

**a) One-to-One**

Each product can have one **detailed info** object.

javascript

CopyEdit

class ProductDetail extends Model {

static fields = {

product: fields.OneToOneField(Product),

manufacturer: fields.CharField({ length: 50 }),

warranty\_months: fields.IntegerField(),

}

}

**Use Case:**  
Each product has exactly **one detail record** with manufacturer info.

**b) Many-to-One**

Many orders can belong to **one user**.

javascript

CopyEdit

class User extends Model {

static fields = {

username: fields.CharField({ length: 50 }),

}

}

class Order extends Model { static fields = {

user: fields.ForeignKey(User),

total: fields.FloatField(),

status: fields.CharField({ length: 20 }),

}

}

**Use Case:**  
A single user can have **multiple orders**.

**c) Many-to-Many**

Products can belong to **multiple categories**, and categories can have **multiple products**.

javascript

CopyEdit

class Category extends Model {

static fields = {

name: fields.CharField({ length: 50 }),

}

}

class Product extends Model {

static fields = {

name: fields.CharField({ length: 100 }),

categories: fields.ManyToManyField(Category),

}

}

**Use Case:**  
You can assign “Electronics” and “Sale” categories to the same product.

**3️⃣ Querying Data**

Your ORM gives **Django-style querying**.

**a) Fetch all objects**

javascript

CopyEdit

const allProducts = await Product.objects.all();

**Use Case:**  
Show all products in your store.

**b) Filter using conditions**

javascript

CopyEdit

const cheapProducts = await Product.objects.filter({ price\_\_lt: 50 });

**Use Case:**  
Show products **cheaper than $50**.

**c) Search text (like SQL LIKE)**

javascript

CopyEdit

const search = await Product.objects.filter({ name\_\_icontains: "phone" });

**Use Case:**  
Show all products with “phone” in the name, ignoring case.

**d) Advanced query with Q objects**

javascript

CopyEdit

const { Q } = require("./orm");

const query = Q.or(

Q.and(

new Q({ price\_\_lt: 50 }),

new Q({ in\_stock: true })

),

new Q({ name\_\_icontains: "special" })

);

const result = await Product.objects.filter(query);

**Use Case:**

* Products **in stock and cheaper than $50**, OR
* Products with **“special”** in the name.

**4️⃣ Creating / Updating / Deleting**

**Create a record**

javascript

CopyEdit

await Product.objects.create({ name: "iPhone", price: 999 });

**Use Case:**  
Add a new product to your store.

**Update a record**

javascript

CopyEdit

await Product.objects.update({ id: 1 }, { price: 899 });

**Use Case:**  
Change the price of product ID 1.

**Delete a record**

javascript

CopyEdit

const product = await Product.objects.get({ id: 2 });

await Product.objects.remove(product);

**Use Case:**  
Remove a discontinued product.

**5️⃣ Aggregates (Calculate numbers)**

**Count**

javascript

CopyEdit

const totalProducts = await Product.objects.count();

**Use Case:**  
Show **total products in store**.

**Min / Max / Avg / Sum**

javascript

CopyEdit

const minPrice = await Product.objects.min("price");

const maxPrice = await Product.objects.max("price");

const avgPrice = await Product.objects.avg("price");

const totalRevenue = await Order.objects.sum("total");

**Use Case:**

* Find **cheapest product**, **most expensive**, **average price**, or **total revenue**.

**6️⃣ Example Workflow in Real Life**

1. Create users:

javascript

CopyEdit

await User.objects.create({ username: "Alice" });

await User.objects.create({ username: "Bob" });

1. Add categories:

javascript

CopyEdit

const electronics = await Category.objects.create({ name: "Electronics" });

const sale = await Category.objects.create({ name: "Sale" });

1. Add products:

javascript

CopyEdit

const phone = await Product.objects.create({ name: "iPhone", price: 999 });

await phone.categories.add(electronics);

await phone.categories.add(sale);

1. Create an order:

javascript

CopyEdit

await Order.objects.create({ user: 1, total: 999, status: "pending" });

1. Search products:

javascript

CopyEdit

const cheapPhones = await Product.objects.filter({ price\_\_lt: 500 });

✅ **Summary:**  
This ORM system lets you:

* Define **fields** (columns) easily
* Create **relationships** between tables
* Query **like Django** (filter, Q, contains)
* Perform **CRUD operations**
* Use **aggregates** like count, min, max, sum