



SDEV 2401

Rapid Backend App Development

Databases, Models, Migrations and the Admin - 1

A LEADING POLYTECHNIC COMMITTED TO YOUR SUCCESS

Expectations - What I expect from you

- No Late Assignments
- No Cheating
- Be a good classmate
- Don't waste your time
- Show up to class

Agenda

On the right is what we will cover today.

- Django ORM: Advanced Filtering and Data Management
- Loading Data with Fixtures
- Defining Models for Filtering
- Creating Views for Advanced Filtering
- Using Q Objects for Complex Queries
- Displaying Filtered Data in Templates
- Challenge
- Summary

Django ORM: Advanced Filtering and Data Management

Django's ORM allows you to perform advanced queries and manage your data efficiently. Let's use a `Product` and `Category` example for an online store.

Loading Data with Fixtures

- You can load initial data into your database using JSON fixtures.
- Example command:

```
python manage.py loaddata products_data.json
```

- This is useful for setting up test data or sharing data between environments.

Defining Models for Filtering

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=8, decimal_places=2)
    category = models.ForeignKey(Category, related_name='products', on_delete=models.CASCADE)
    in_stock = models.BooleanField(default=True)

    def __str__(self):
        return self.name
```

Creating Views for Advanced Filtering

You can filter products by category, price, or stock status in your views.

```
from django.shortcuts import render, get_object_or_404
from .models import Product, Category

def product_list(request):
    products = Product.objects.filter(in_stock=True)
    return render(request, 'store/product_list.html', {'products': products})

def category_products(request, category_id):
    category = get_object_or_404(Category, id=category_id)
    products = category.products.filter(price_lt=50)
    return render(request, 'store/category_products.html', {'category': category, 'products': products})
```

Using Q Objects for Complex Queries

Q objects allow you to combine filters with OR and AND logic.

```
from django.db.models import Q

def search_products(request):
    query = request.GET.get('q', '')
    if query:
        products = Product.objects.filter(
            Q(name__icontains=query) | Q(category__name__icontains=query)
        )
    else:
        products = Product.objects.none()
    return render(request, 'store/search_results.html', {'products': products, 'query': query})
```

Displaying Filtered Data in Templates

```
<!-- store/product_list.html -->
<ul>
  {% for product in products %}
    <li>
      <strong>{{ product.name }}</strong> - ${{ product.price }}
      <span>{{ product.category.name }}</span>
      {% if product.in_stock %}
        <span class="text-green-600">In Stock</span>
      {% else %}
        <span class="text-red-600">Out of Stock</span>
      {% endif %}
    </li>
  {% endfor %}
</ul>
```

Summary

- Use fixtures and `loaddata` to quickly populate your database.
- Filter and search your data using Django ORM's powerful query methods.
- Use Q objects for complex queries.
- Display filtered results in your templates for a dynamic user experience.



Example

Let's go do an example together