From Data to Display: Rendering a Dynamic List in Django

A step-by-step guide to connecting your database to your HTML templates.

Part 03 of the Django for Beginners Series

The Disconnect: Backend Data vs. Frontend Display

Our Data

```
obj1: { title: 'Movie A', year: 2022, … }
```

```
obj2: { title: 'Movie B', year: 2023, … }
```

```
obj3: { title: 'Movie C', year: 2021, … }
```

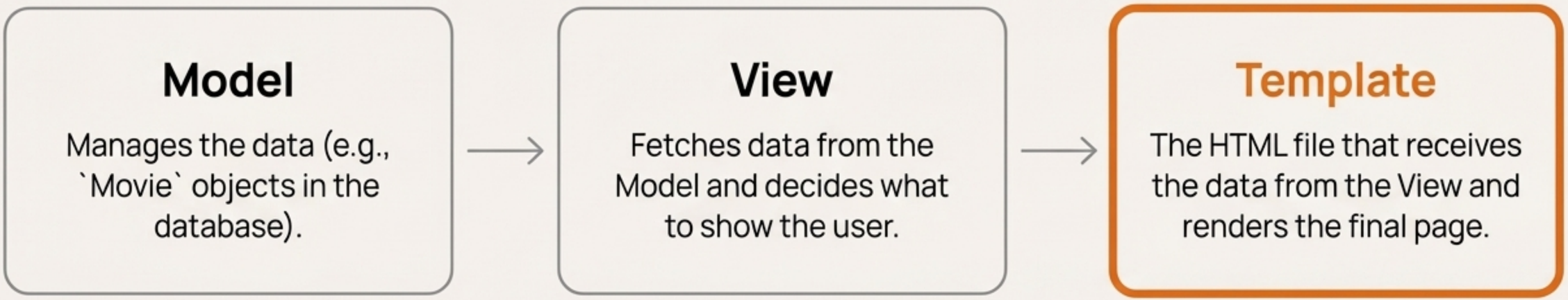
Our Goal (Static HTML)

```
<ul>
  <li>Movie A - 2022</li>
  <li>Movie B - 2023</li>
  <li>Movie C - 2021</li>
</ul>
```

How do we build this list dynamically, so it updates automatically when our data changes?

The Django Way: Separating Logic from Presentation

Django uses the **Model-View-Template (MVT)** architecture to keep code organized and maintainable.



Our task is to connect the **View** (Python logic) to the **Template** (HTML presentation).

Step 1: Create the `templates` Directory

Django automatically looks for a directory named `templates` within your app to find HTML files.

```
my_app/
├── migrations/
├── __init__.py
├── admin.py
├── apps.py
├── models.py
├── tests.py
├── views.py
└── templates/ <— CREATE THIS FOLDER
```

Pro Tip: This convention is configurable in `settings.py`, but following the default is best practice for beginners.

Step 2: Build the Static HTML Blueprint

Inside your new `templates` folder, create an HTML file. Let's call it `index.html`. This file will serve as the basic structure for our movie list.

```
<!DOCTYPE html>
<html>
<head>
    <title>Movie List</title>
</head>
<body>
    <h1>All Movies</h1>

    <!— We will dynamically generate this list —>
    <div>
        <h2>Movie Title</h2>
        <p>Summary of the movie...</p>
        <p>Year: ###</p>
    </div>
</body>
</html>
```

Step 3: Connect the View to the Template

views.py

We use Django's `render()` shortcut to combine a template with a data context (which we'll add next) and return a complete `HttpResponse`.

Before

```
from django.http import HttpResponse

def movie_list(request):
    return HttpResponse("Hello, world. ...")
```

After

```
from django.shortcuts import render # Import render

def movie_list(request):
    # We will fetch data here soon
    return render(request, 'index.html') # Render the template
```

Django searches for this file inside any `templates` directory it finds.

Step 4: Fetch All Movie Objects in the View

views.py

We use our `Movie` model to query the database. The `.objects.all()` method retrieves every record from the movie table.

```
from django.shortcuts import render
from .models import Movie # 1. Import the Movie model

def movie_list(request):
    # 2. Query the database for all Movie objects
    movies = Movie.objects.all()

    return render(request, 'index.html')
```

1

2

The variable `movies` now holds a 'QuerySet'—a list-like object containing all our movie instances.

Step 5: Pass Data to the Template via a Context Dictionary

views.py

The `context` is a Python dictionary where keys become variable names in the **template** and **values** are the data they represent.

```
from django.shortcuts import render
from .models import Movie

def movie_list(request):
    movies = Movie.objects.all()

    # 3. Create the context dictionary
    context = {
        'movie_list': movies
    }

    return render(request, 'index.html', context) # 4. Pass context to render
```

3

4

This key is now a variable named `movie_list` available inside `index.html`.

views.py

Code Reference: The Complete View Function

Here is the final code for `views.py`. It performs three key actions: fetching data, packaging it, and rendering the template.

```python
# views.py

from django.shortcuts import render
from .models import Movie

def movie_list(request):
    # 1. FETCH DATA
    # Use the model to get all movies from the database.
    movies = Movie.objects.all()

    # 2. PACKAGE DATA
    # Create a context dictionary to send to the template.
    # The key 'movie_list' will be the variable name in HTML.
    context = {
        'movie_list': movies
    }

    # 3. RENDER RESPONSE
    # Combine the request, the template file, and the context
    # to generate the final HTML.
    return render(request, 'index.html', context)
```

The Bridge to Dynamic HTML: Django Template Language (DTL)

DTL is a simple language for embedding logic into your HTML. It's designed to be designer-friendly and avoids executing arbitrary Python code.

`{{ variable }}`

Purpose

For printing variables. Accesses dictionary keys or object attributes.

Example

```
<{{ movie_obj.title }}>
```

`{% tag %}`

Purpose

For logic like loops and conditionals.

Example

```
{% for movie in movie_list %}
```

We will use a `for` tag to loop through our `movie_list` and variable tags to display each movie's details.

index.html

Step 6: Iterate Through Data with a `for` Loop

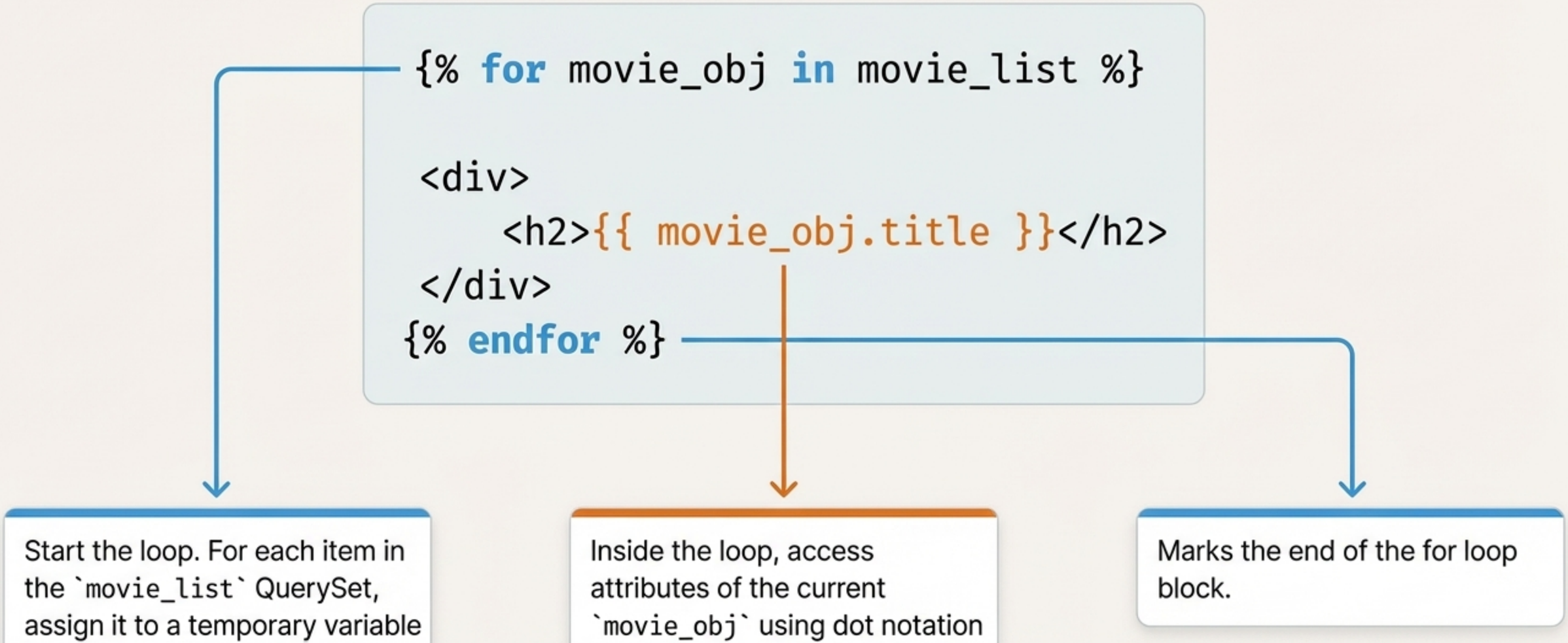
Replace the static placeholder `div` with a DTL `for` loop. This loop will execute once for every movie object in the `movie_list` we passed from the view.

```
<h1>All Movies</h1>

{% for movie_obj in movie_list %}
    <div>
        <h2>{{ movie_obj.title }}</h2>
        <p>{{ movie_obj.summary }}</p>
        <p>Year: {{ movie_obj.year }}</p>
    </div>
{% endfor %}
```

This is the same variable name we defined as a key in our context dictionary in `views.py`.

Anatomy of the DTL `for` Loop



index.html

Code Reference: The Complete Dynamic Template

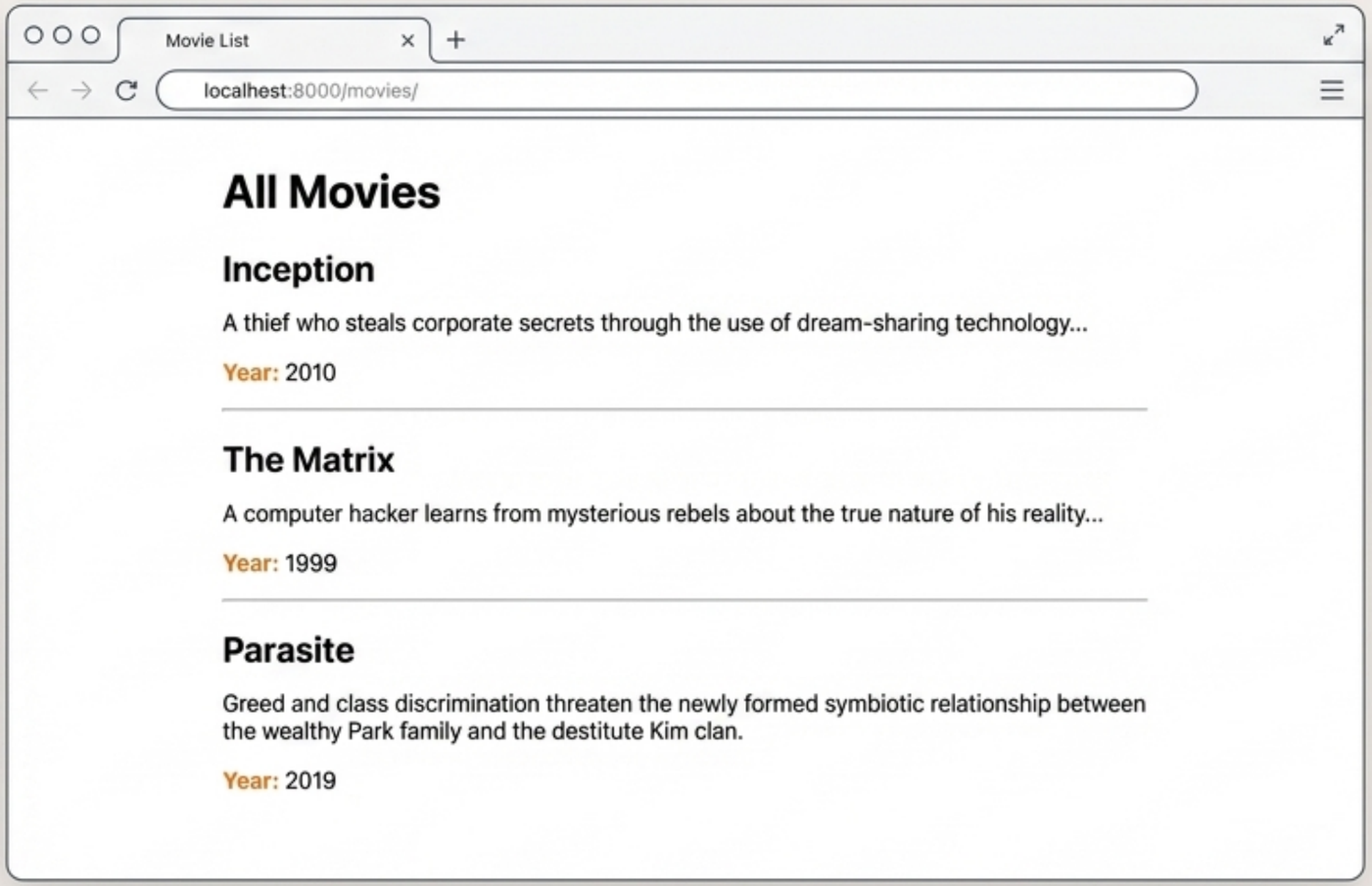
Your final `index.html` should now look like this. It is no longer a static file, but a true template ready to display any number of movies.

```
<!DOCTYPE html>
<html>
  <head>
      <title>Movie List</title>
  </head>
  <body>
      <h1>All Movies</h1>

      {% for movie_obj in movie_list %}
          <div style="border-bottom: 1px solid #ccc; padding-bottom: 1em; margin-bottom: 1em;">
              <h2>{{ movie_obj.title }}</h2>
              <p>{{ movie_obj.summary }}</p>
              <p><strong>Year:</strong> {{ movie_obj.year }}</p>
          </div>
      {% endfor %}
  </body>
</html>
```

The Result: A Fully Rendered, Dynamic Page

After saving your `views.py` and `index.html` files, refresh the page in your browser. Django will now execute the view, run the template logic, and render a complete list of all movies from your database.



From Data to Display: Key Takeaways

The Workflow

1. **Structure**
   Create a `templates` folder inside your app.
2. **View Logic (`views.py`)**
   - Fetch data from your models (e.g., `Movie.objects.all()`).
   - Create a `context` dictionary to pass the data.
   - Use `render(request, 'template.html', context)`.
3. **Template Logic (`.html`)**
   - Use a `{% for ... %}` loop to iterate over the data.
   - Use `{{ variable.attribute }}` to display the data points.

Core Principle: This process successfully separates your Python logic from your presentation HTML, leading to cleaner, more scalable applications.