Theory: Using models with templates

29 minutes 0 / 5 problems solved Skip this topic Start practicing

565 users solved this topic. Latest completion was about 6 hours ago.

Current topic:

Using models with templates

Topic depends on:

- HTML forms (Stage 4)
- Audio
- Django template language (Stage 2)
- Django ORM

Table of contents:

↑ Using models with templates
§1. Passing Object to Template
§2. Forms
§3. Many Objects
§4. Using Filters
§5. Conclusion
Feedback & Comments

The advantage of using a framework is that all its parts are easily combinable. We have *templates* to render HTML pages and *models* to work with the database. If you are entertaining the idea of using them together, you're right to do so.

This time we'll focus not only on simple HTML layout, but on *forms* that are used for communication with the server and sending data to it.

§1. Passing Object to Template

Do you like music? Do you like it as much as creating your own music service? With this service you'll be able to listen to a variety of tracks, search for specific songs and mark your favorite ones. We don't need much for the first prototype – let's use Django's `User` model and define another one ourselves:

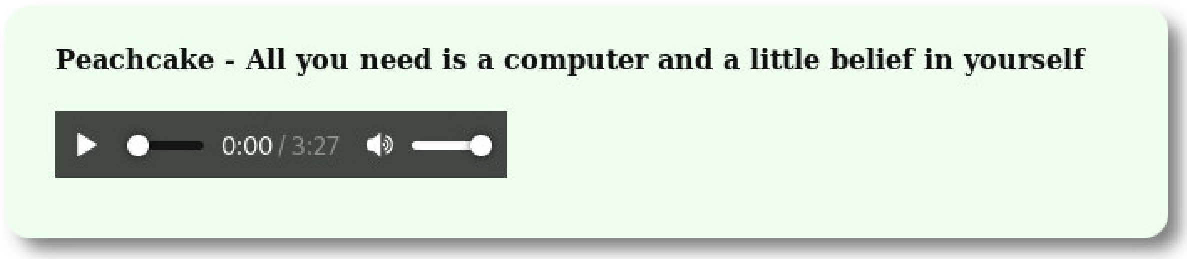
```
from django.contrib.auth.models import User
from django.db import models


class Song(models.Model):
    title = models.CharField(max_length=128)
    artist = models.CharField(max_length=128)
    path_to_file = models.FileField(upload_to='static/')

favorite_by = models.ManyToManyField(User, related_name='favorite_songs')
```

We can pass a context dictionary to the template to fill it with data. Assume that we have an instance of `Song` class in a `song` variable. In this case our context dictionary will be simply `{'song': song}`. In models, all fields will be available for template layout; you can even access foreign keys and their fields.

Using fancy CSS for our prototype seems a tinge unnecessary, so let's create a very moderate HTML template:

```
<h4>{{ song.artist }} - {{ song.title }}</h4>
<audio controls>
    <source src="/static/{{ song.path_to_file }}" type="audio/mp3">
</audio>
```

The result looks like this:



Peachcake - All you need is a computer and a little belief in yourself

All attributes of models are accessed through the dot. The `path_to_file` is a relative path to a file from `upload_to` directory we choose in the model. We don't redefine the default value for static files in *settings.py* module and use the "*static*" folder for it as a file source prefix.

To make the static files directory work, you should first create it in the root of your project and then define STATICFILES_DIRS in the *settings.py* module.

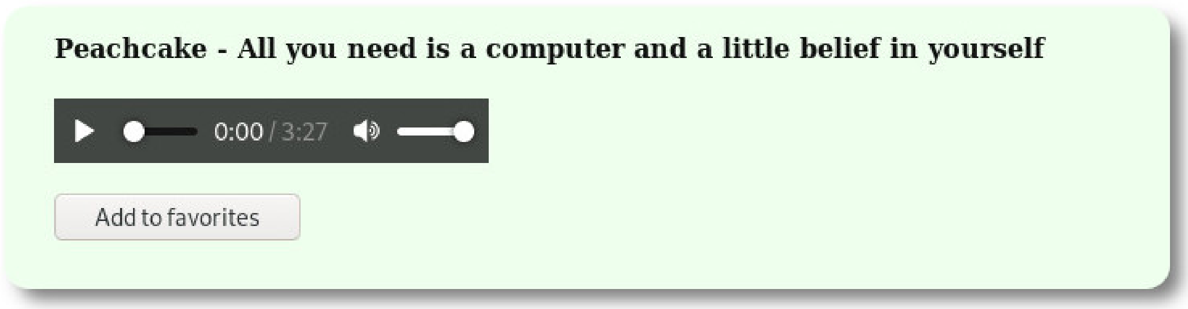
§2. Forms

One of the best features of music services is the possibility to create playlists with favorite songs; our service definitely needs that! We extend the existing template with a *form* to use it in *POST* requests:

```
<form action="/add_to_favorites" method="post">
   {% csrf_token %}
   <input type="hidden" name="song_id" value="{{ song.id }}">
   <button type="submit">Add to favorites</button>
</form>
```

We access the `id` field of a variable through the dot. If you looked closely at the code, you surely noticed we're also using a cryptic tag `{% csrf_token %}`. CSRF is an abbreviation for [Cross-Site Request Forgery](). We don't want any fraud action to happen, so in forms we must always use this tag to secure our applications. CSRF token is a generated sequence of symbols that the server uses to identify a user's session. If the sequence matches, the form is considered reliable.

Now we have a form to send to the server; it will process the request and add the song of our choice to favorites.


Peachcake - All you need is a computer and a little belief in yourself

Your request will work only if your application has a POST handler for the address "*/add_to_favorites*". You can find this URL in the action field of the form.

§3. Many Objects

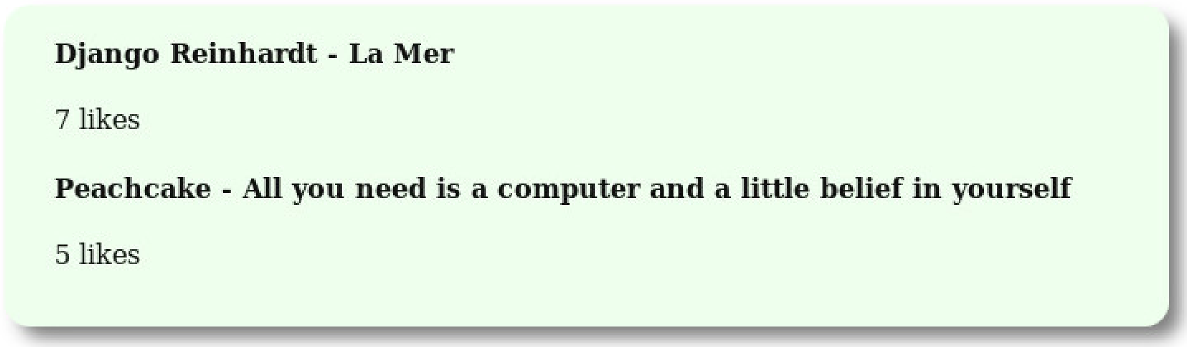
A whole playlist will be shown on the main page, and here it's slightly different from having just one object per page. Using a `for` loop and a QuerySet would work well for a situation like that.

We create a new page and change the context data passed to the template. Ten or fewer random songs from the database will appear; now our context data are `{'songs': Song.objects.all()[:10]}`:

```
{% for song in songs %}
<div>
   <h4>{{ song.artist }} - {{ song.title }}</h4>
   <div>{{ song.favorite_by.count }} likes</div>
</div>
{% endfor %}
```

Why stop here? Let's add another great feature to our server and for each song display the number of users who added it to their favorites. To do this, access a user's QuerySet with `favorite_by` attribute defined in the model and call the `count` method on it. As for other Python objects, the only candidates for templates are methods with no parameters; `count` is one of them.

Now the users can see which songs are really popping and rocking!


Django Reinhardt - La Mer

7 likes

Peachcake - All you need is a computer and a little belief in yourself

5 likes

It's usually better to calculate such data at a stage of processing the request in a handler. We should define only the representation of a page in our template.

§4. Using Filters

Django provides filters for processing values in templates. Sometimes they can really save us time, yet they may also yield surprising results. The template processor uses the string representation of an object, which has its pros and cons. Being careful with that is the key to getting the most of Django.

Let's slightly change the template from the previous example:

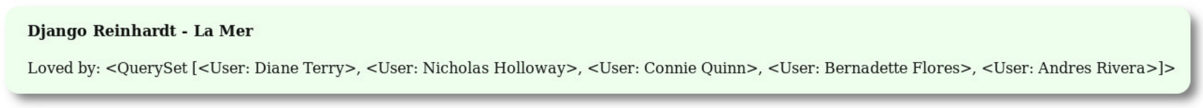
```
{% for song in songs %}
<div>
   <h4>{{ song.artist }} - {{ song.title }}</h4>
   <div>Loved by {{ song.favorite_by.all|random }}</div>
</div>
{% endfor %}
```

We render a random `User` who liked a song (provided we have the user's consent, of course). Upon receiving the QuerySet, we pass it to the `random` filter to get one username to show.

Though this feature is quite experimental, it still has a chance to win the hearts of our users, so let's render five of them instead of just one:

```
{% for song in songs %}
<div>
   <h4>{{ song.artist }} - {{ song.title }}</h4>
   <div>Loved by: {{ song.favorite_by.all|slice:5 }}</div>
</div>
{% endfor %}
```

We sliced the QuerySet and limited its length to five items: what could go wrong? Yet when we open the browser, we get a strange output:

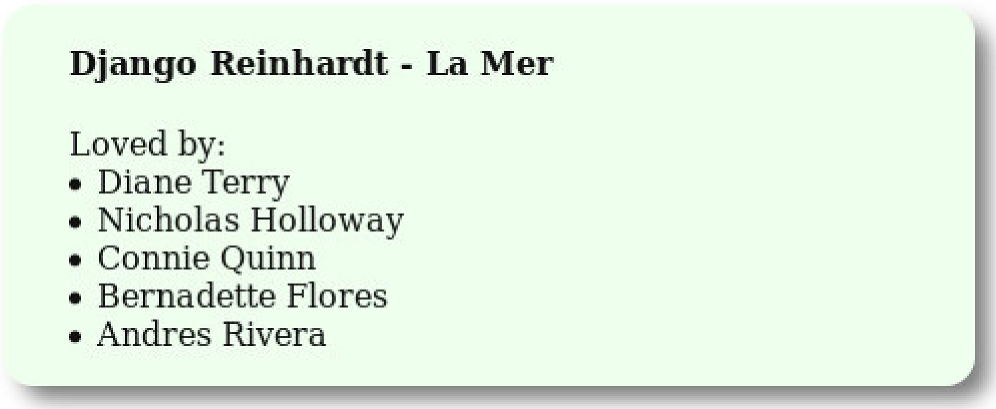


```
Django Reinhardt - La Mer
Loved by: <QuerySet [<User: Diane Terry>, <User: Nicholas Holloway>, <User: Connie Quinn>, <User: Bernadette Flores>, <User: Andres Rivera>]>
```

What kind of English is that? It seems like Django converted the QuerySet to a string and rendered it to HTML. It's not a result to be particularly proud of. Don't lose heart though as we have another trick up our sleeve:

```
{% for song in songs %}
<div>
   <h4>{{ song.artist }} - {{ song.title }}</h4>
   <div>Loved by: {{ song.favorite_by.all|slice:5|unordered_list }}</div>
</div>
{% endfor %}
```

This time we prudently convert all values to an HTML unordered list with the `unordered_list` filter, and our output now looks quite satisfying:



```
Django Reinhardt - La Mer

Loved by:
• Diane Terry
• Nicholas Holloway
• Connie Quinn
• Bernadette Flores
• Andres Rivera
```

§5. Conclusion

There are ways to take this idea further and really elaborate: for example, one could think of the best and most convenient design for the users. You could think about that too, or make your own application and share your skills and the fruit of your labor with your friends. The main thing is – you know enough to start making pages using Django models.

Report a typo

48 users liked this theory. 5 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (10) Hints (0) Useful links (0) Show discussion