Theory: Submitting data

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

745 users solved this topic. Latest completion was about 5 hours ago.

Making a service *interactive* implies allowing your users to receive and send information. We use *GET* requests to get a page and *POST, PUT, DELETE* requests to update data on the server side. It's not a requirement, but a strong rule to follow. You can still make a valid application if you break this rule, but its behavior would be unexpected for its users and developers alike.

Let's look closer at how we can exchange data and make the first *POST* and *DELETE* request handlers!

§1. Query Parameters

I have a dream. In fact, all of us do, and sharing dreams and desires with others is a thing. Maybe if you share your wish with a right person, they can make it come true? Creating a *wishlist* is a simple way to do that. Ideally, the service should allow users to search wishes by keywords, create new wishes and delete those that are not relevant for them anymore.

We want the service to do quite a variety of actions, so how do you tell apart these different? Would their only distinctive feature be their HTTP methods? Well, to start processing any request we should definitely look at its method. The URL for different types may stay the same while the action of the server would be different. We use only one URL for each user: *<address of server>/<username>*. If the hostname is "*www.makeawish.happen*" and the username is "*Mymble*", the full address to Mymble's wishlist will be "*www.makeawish.happen/Mymble*".

To search, we would use a *GET* request. For example, searching by keyword may look like *www.makeawish.happen/Mymble?q=cake.* We use a query string after the question mark to specify the parameters and their values. To access these values in a request handler, we can use the GET attribute of `HttpRequest` class. Query parameters are associated with *GET* requests so we can find them in this attribute.

The "*q*" parameter in the `GET` attribute can now be accessed as a key in a dictionary because this attribute is a `QueryDict` instance. QueryDict is a subclass of Python's dictionary, but the main difference is that `QueryDict` is immutable – that is, you can't modify the user's request, which makes sense.

§2. Request Body

All requests have a body. Sometimes it's empty, but for *POST* requests it's filled with data. In these cases you can find what the client sent to the server in the POST attribute of `HttpRequest` class.

How does a client create the request body? You cannot find it in the URL, but you can pass it in an HTML form. Assume we have a form:

```
<form action="/Mymble" method="post">{% csrf_token %}
  <input name="wish">
  <button type="submit">Add a wish</button>
</form>
```

The "*wish*" parameter is stored in the `POST` attribute. This attribute is also a `QueryDict` instance, so use it as a usual dictionary; still, do not change anything in it or else you will get an `AttributeError` exception.

Let us now move on to making a handler and processing a new wish.

> We have only *GET* and *POST* attributes in the `HttpRequest` class. Even though we have other HTTP methods, there are no other attributes with their names in an instance of this class.

Current topic:

Stage 4 Submitting data

Topic depends on:

- ✗ Stage 4 Domains
- ✗ Stage 4 HTML forms
- ✗ Stage 1 Processing requests

Topic is required for:

Forms and validation

Table of contents:

§3. POST request

Making a wish is the first step of bringing it to life. We use the *POST* handler to save a wish in the application; the signature of a method is similar to the *GET* handler. The lists are stored only in a static attribute of a class – let's see an example:

```
from collections import defaultdict

from django.shortcuts import redirect
from django.views import View


class WishListView(View):
    wishlist = defaultdict(list)


    def post(self, request, *args, **kwargs):

        author = request.user.username if request.user else 'Anonymous'

        wish = request.POST.get('wish')

        self.wishlist[author].append(wish)

        return redirect('/')
```

The wishes are grouped by the author; if a user is not authorized, their wish goes to the '*Anonymous*' group. When a wish is saved, the handler redirects the client to the "*/*" page, which is usually the main page of the application. Redirection is a mechanism that prevents double sending of modification requests. Assume the user sends data and refreshes the page the very next moment. Refreshing repeats the action and sends data again. It's not a tragedy to save the wish twice, but how about spending the user's money twice or thrice? Will the user like that? If a request is redirected, it's not possible to repeat the previous action, so let's play it safe here.

Django matches the *HTTP* method with the same-name method in a `View` class. With this type of inheritance, only the implemented methods are allowed in requests. We do not have ***a** GET* method for a given link and when a client tries to make this request, the answer will be "*405 Method Not Allowed*".

§4. DELETE request

Fleeting wishes come and go; and when they go, our user would probably want to remove them from the wishlist. The most appropriate method for this task is *DELETE*.

We add the `delete` handler to the same class and use the same link for different requests. As we said, it's valid to make requests with different HTTP methods to the same link, and the type of method defines which handler will process the request.

```python
from collections import defaultdict

from django.shortcuts import redirect
from django.views import View


class WishListView(View):
    wishlist = defaultdict(list)


    def delete(self, request, wish, *args, **kwargs):

        author = request.user.username if request.user else 'Anonymous'

        if wish in self.wishlist[author]:

            self.wishlist[author].remove(wish)

        return redirect('/')



    def post(self, request, *args, **kwargs):

        ...
```

We cannot use a `POST` attribute of a `HttpRequest` instance: it contains data only for *POST* requests, so we pass a wish in a query to define what to remove.

You can define the named parameters like one above with urlpatterns.

In the end, the user is redirected again, but this time it's not obligatory. Think about it: it's impossible to delete the same object twice, so it's safe to call this method several times in a row as the object is already removed after the first call. The property of an operation to be applied several times without changing the result beyond the initial call is called **idempotence**.

DELETE, PUT, GET methods are idempotent while *POST* is not.

§5. Conclusion

If you didn't make your wishlist yet, it might be interesting to do so, but you can also go ahead and make an entire wishlist service for you and other folks out there. You can customize the pages, implement all *GET* methods and invite your friends to post their wishes and share them with the world (wide web).

Report a typo

46 users liked this theory. **3** didn't like it. **What about you?**

Start practicing

Comments (5) Hints (0) Useful links (0) Show discussion