CRUD Operations and Forms

In the present document we will adapt our forms from the previous document in order to accept Create, Read, Update, and Delete (CRUD) operations that affect the database.

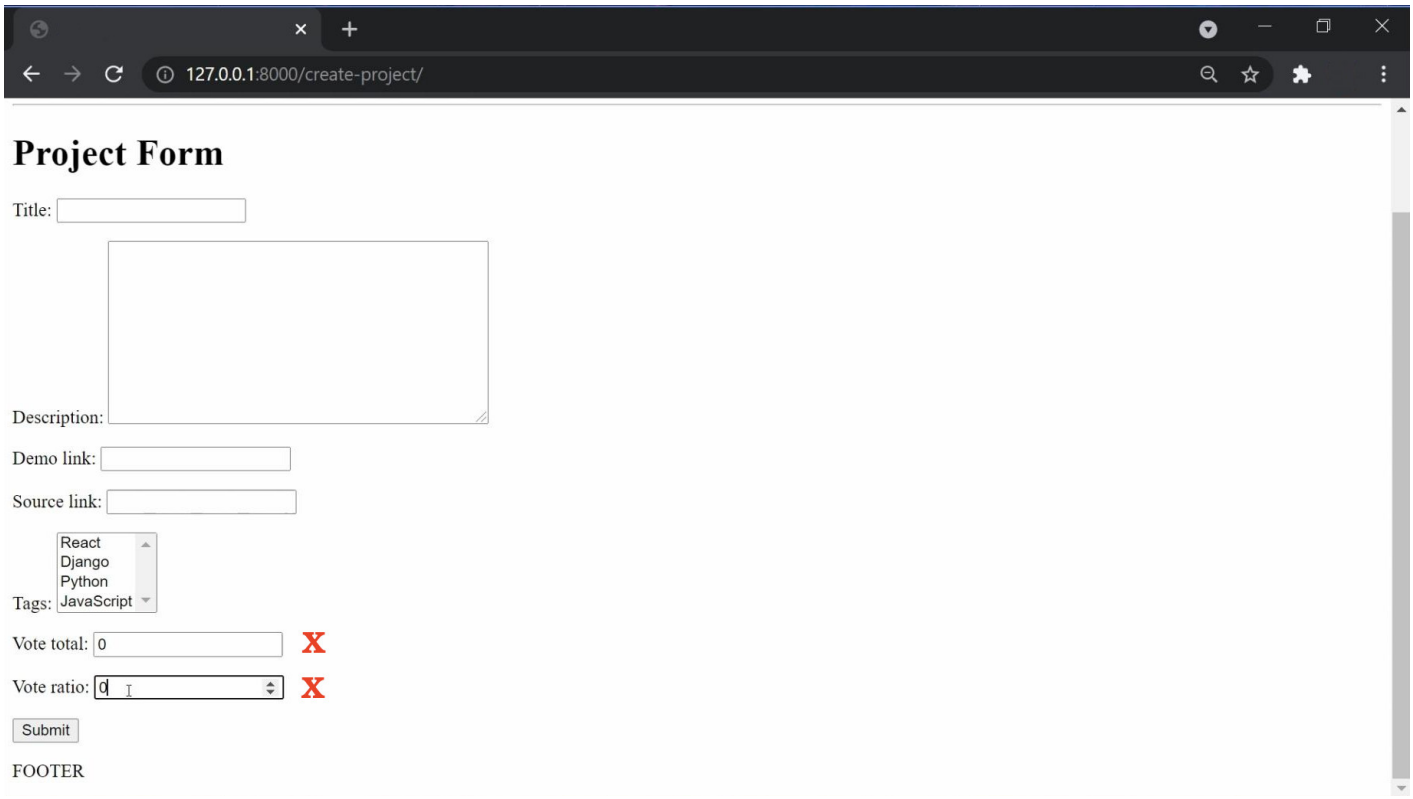
There are four verbs used in any web operation defined in the HTTP/HTTPS protocol:

Verb	Description	Database Equivalent	Operation
POST	Informs the site that there's data being sent to it.	INSERT	CREATE
GET	Informs the site that should send a response with the data indicated by the argument	SELECT	READ
UPDATE	Informs the site that the data sent must update already existing data.	UPDATE	UPDATE
DELETE	Informs the site that it should any data indicated by the parameter	DELETE	DELETE

We need to modify our form views in order to respond to each one of the HTTP/HTTPS verbs and interact with our database accordingly.

Initial Modifications to our Forms

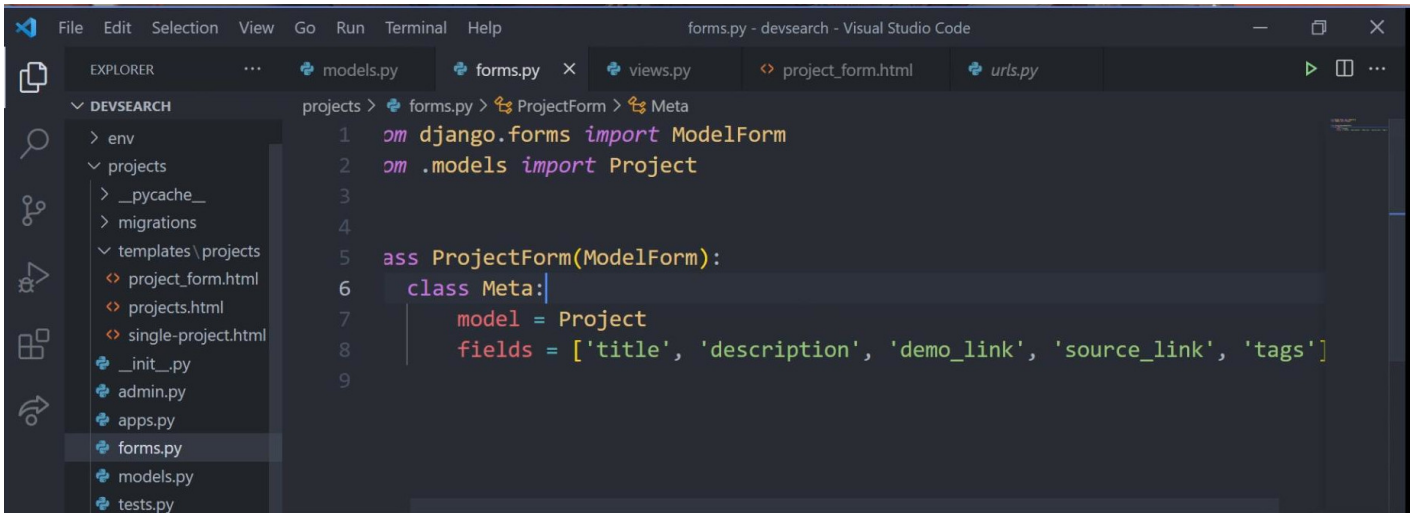
First, we are not interested in showing all the fields present in our /projects/models.py



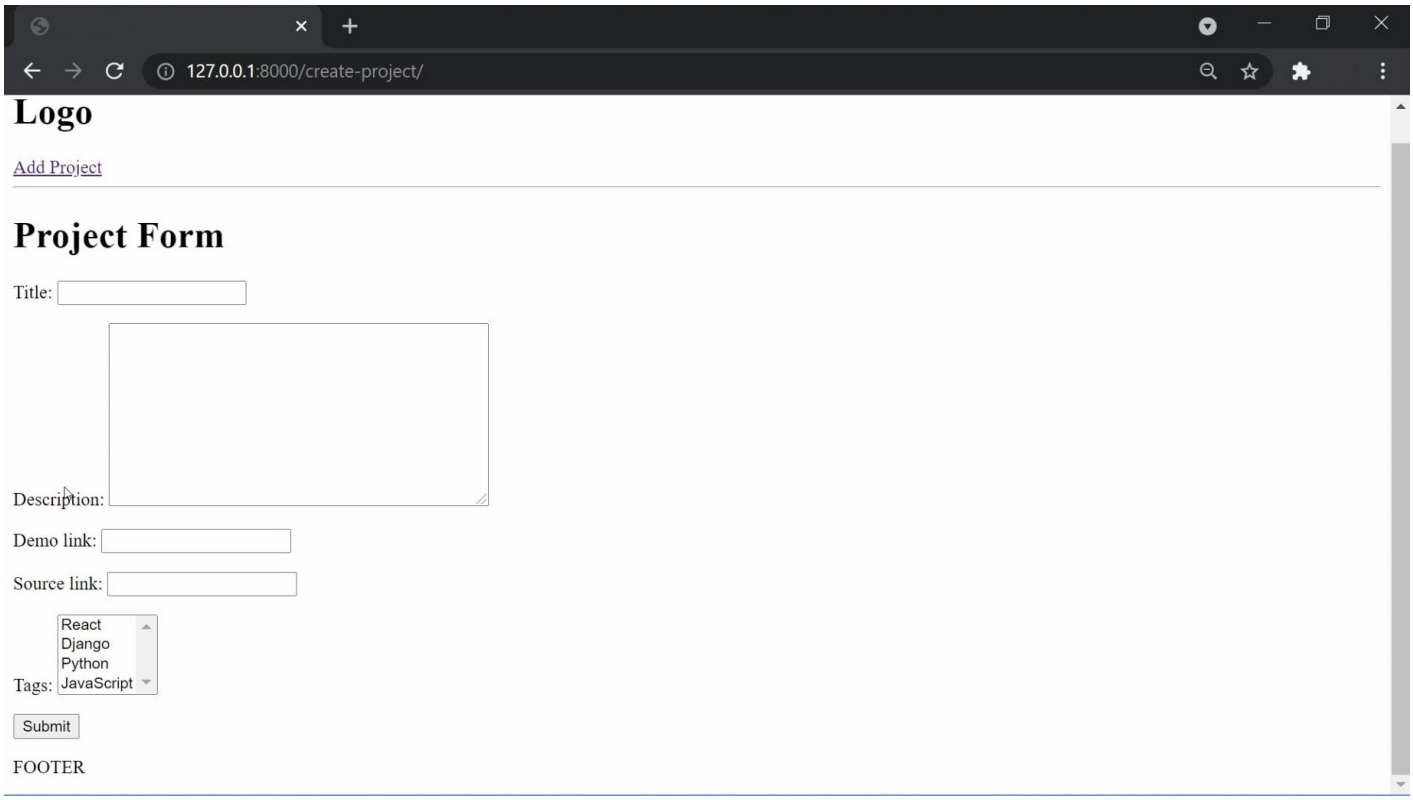
We want to leave Vote total and Vote ratio out of the form. With that in mind, let's modify our forms.py in /projects/forms.py to accept only the fields for:

- title
- description
- demo_link
- source_link
- tags

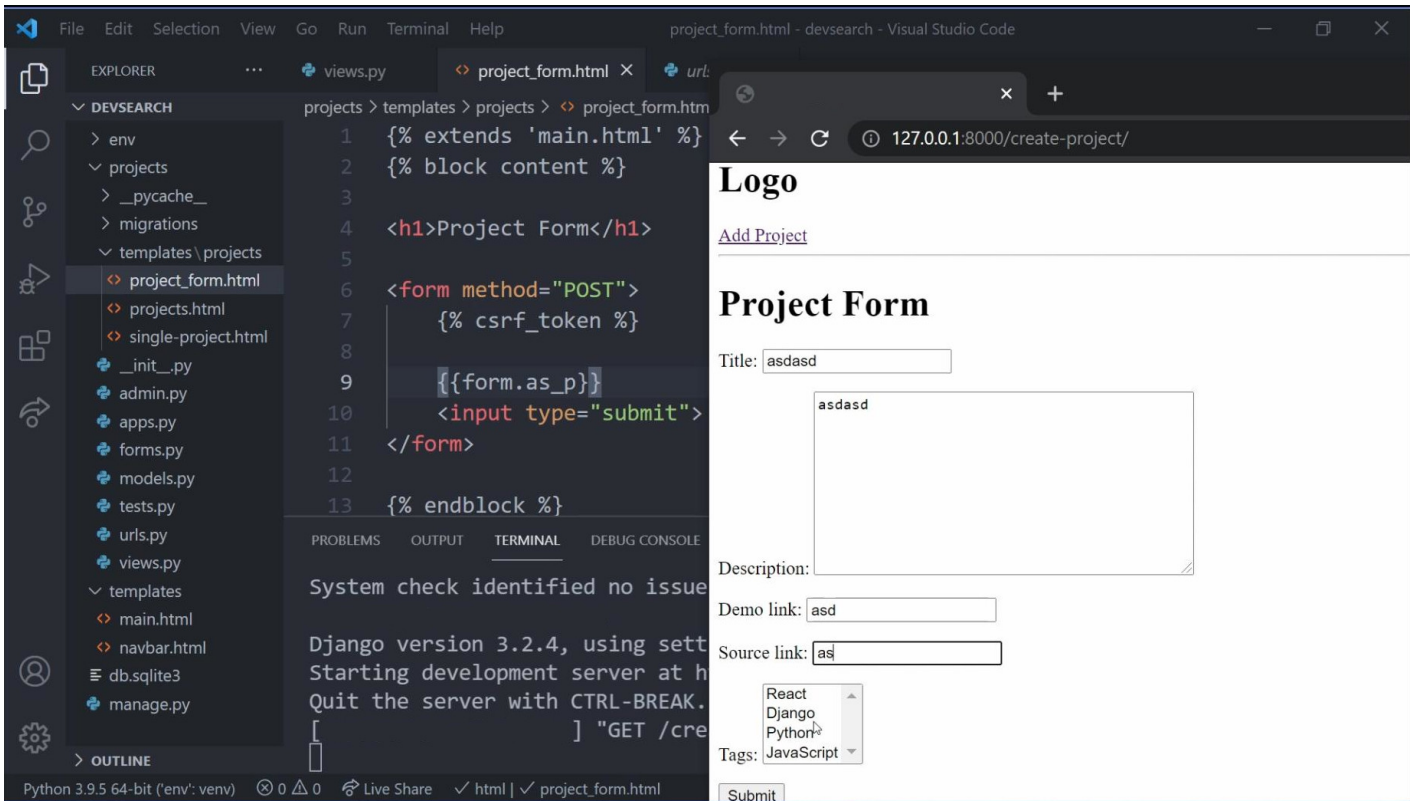
in our createProject class.



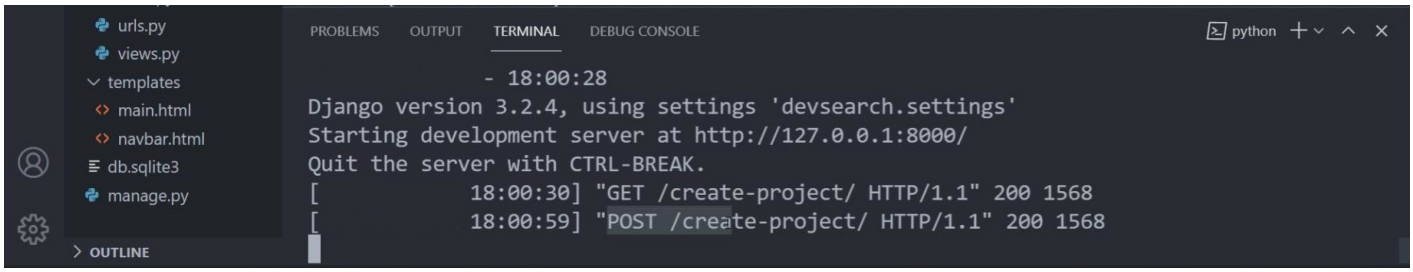
so, our form should look like this:



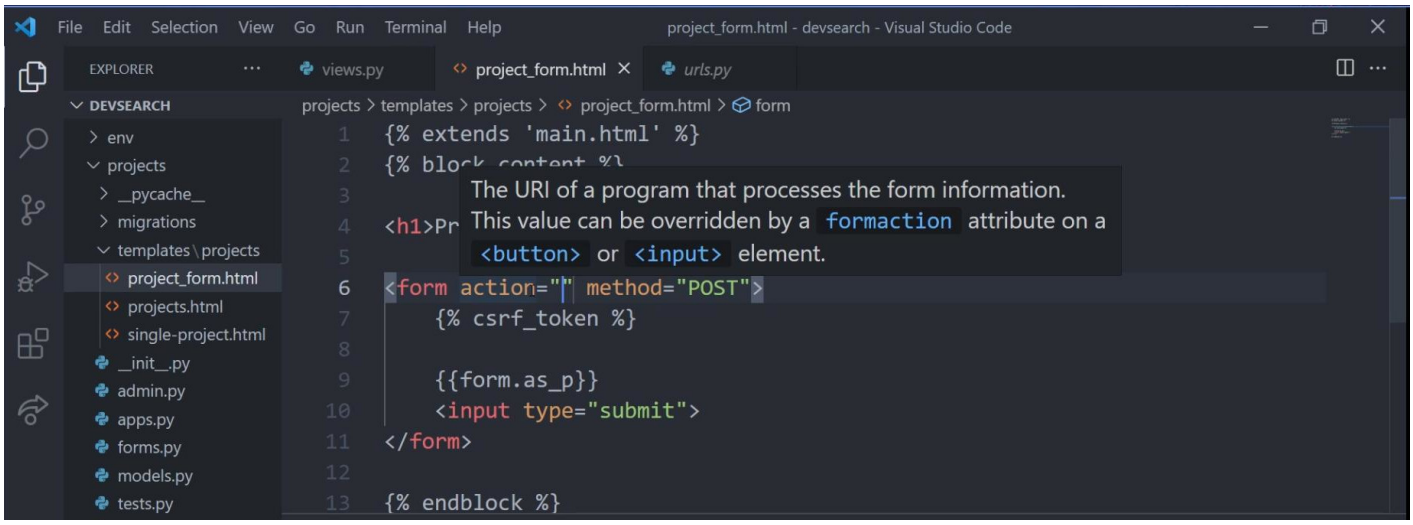
If you submit something into the form:



you should be able to see the POST method after the submission:



The reply for the POST method is given by the form argument method in the project_form.html, but the form doesn't do anything with the input data. The reason is that there's another argument missing:



This missing argument is the formaction attribute. The formaction attribute is the direction to the function that will process the form data.

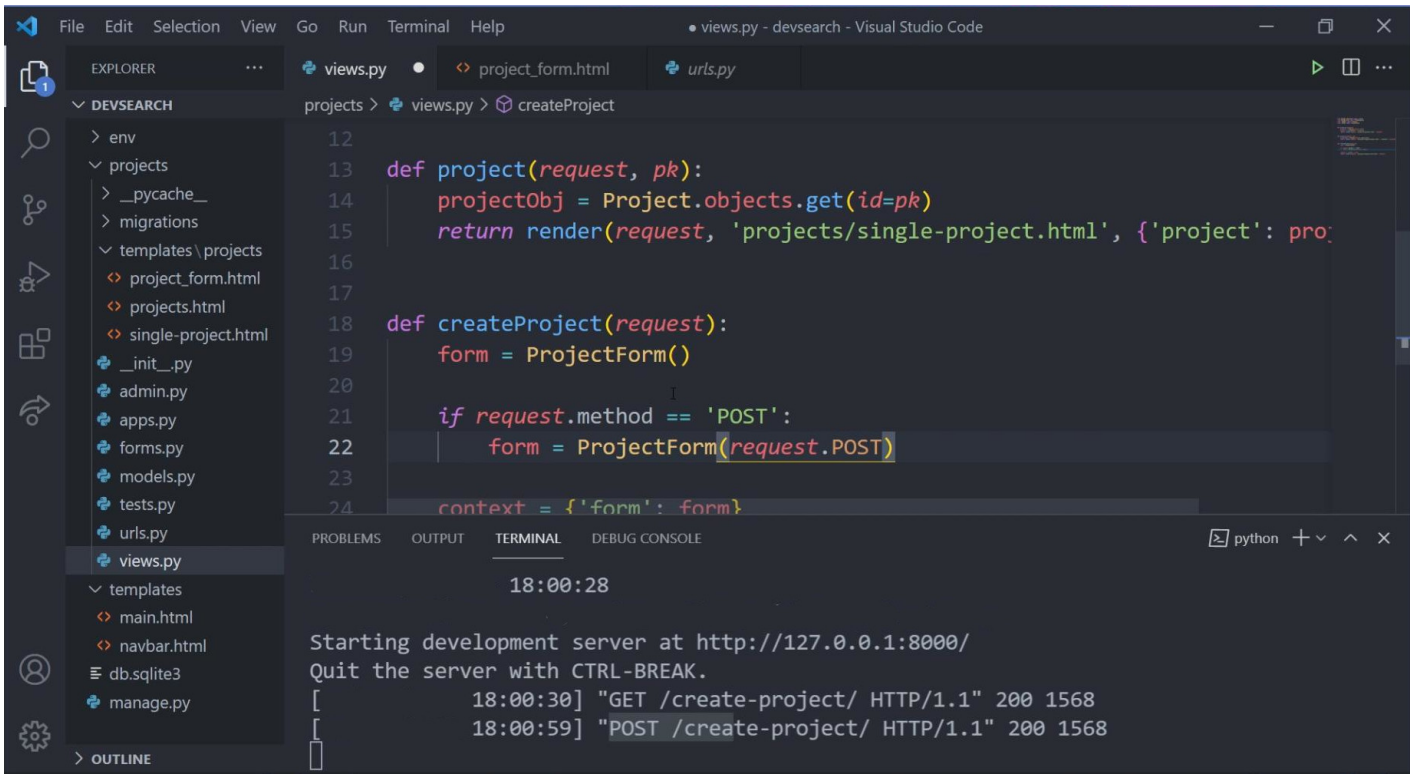
For the time being, we'll leave this argument empty (an empty string) since we have to define the processing function first. We'll return after we have defined such function.

Let's see how the data is passed from the html form to the function form createProject class. Our first order of business is to instanciate our ProjectForm class inside the createProject class in the /projects/views.py

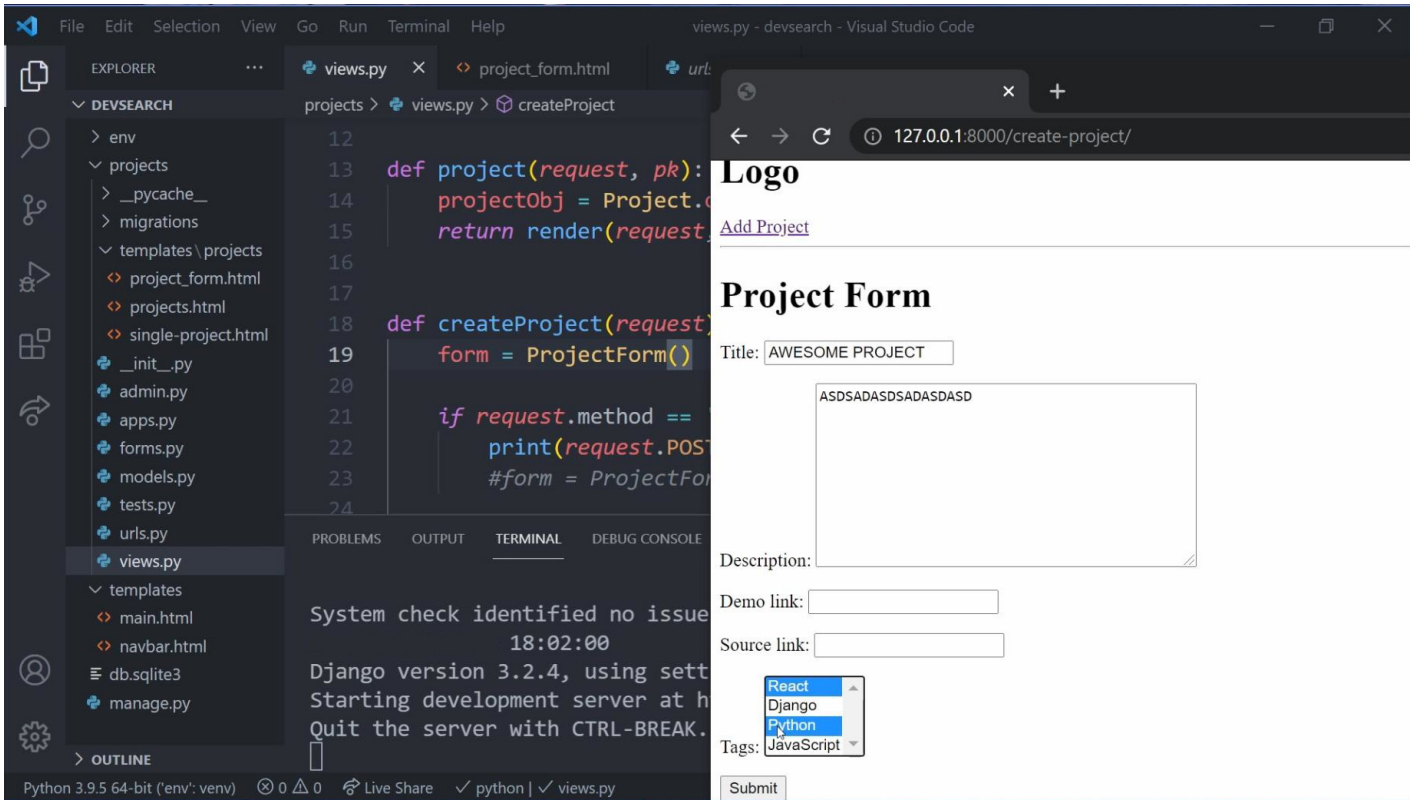
The *request* parameter contains information on the request type. The request type is identified by an enumerate:

```
request.POST
request.GET
request.UPDATE
request.DELETE
```

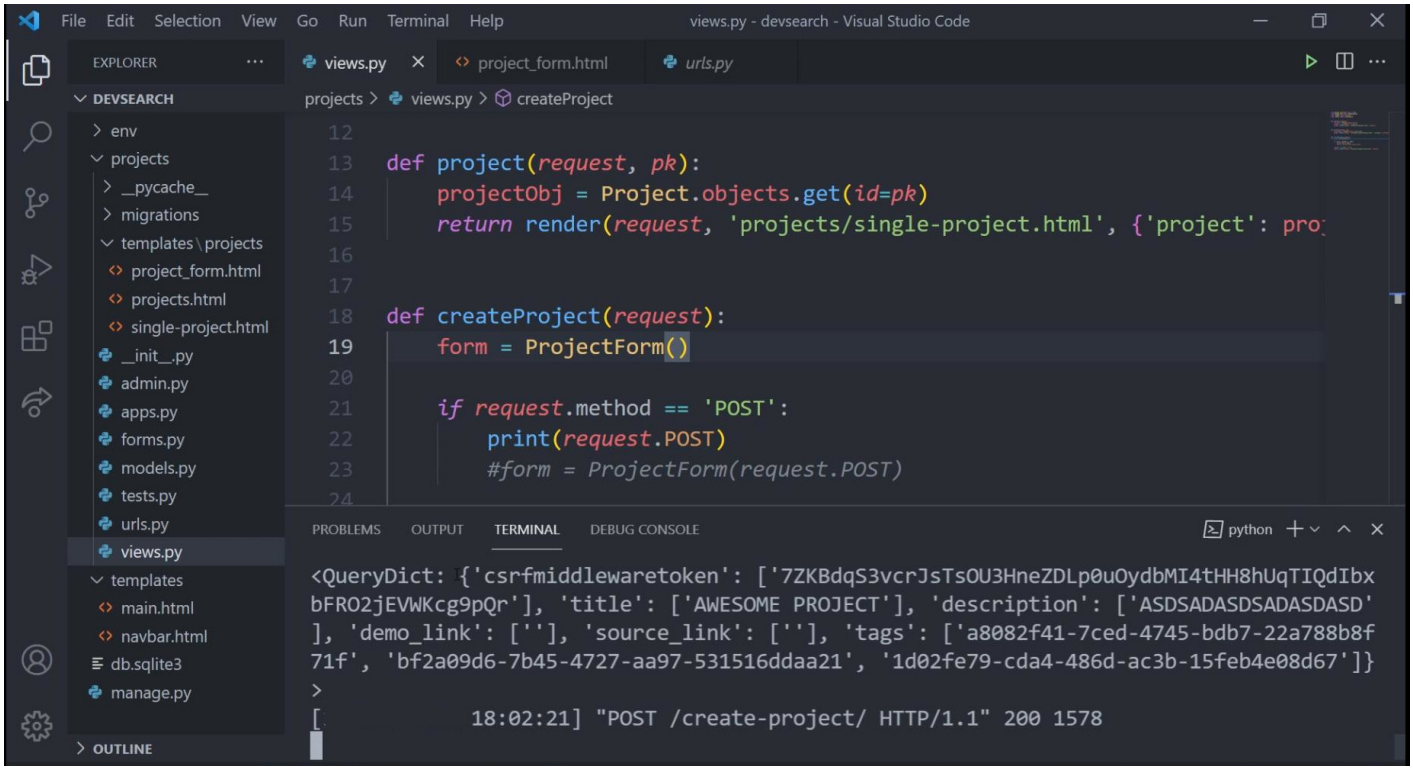
that we can use to redirect the function accordingly. In our case, we'll use *request.POST* and retreive the form input data on a variable *form*



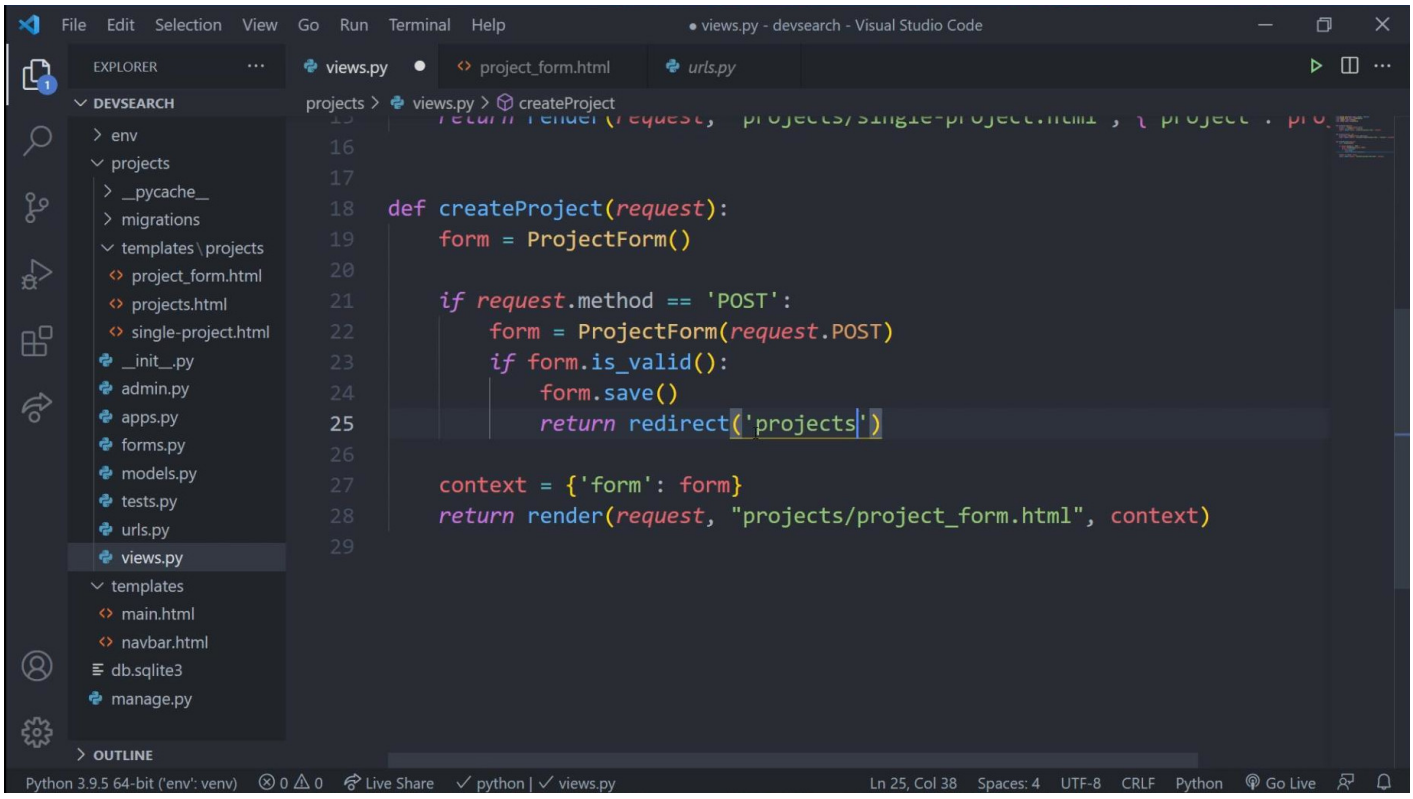
But for demonstration purposes, let's comment the line which assigns the variable from inside the ***if*** statement and print the argument *request.POST* and introduce some data into the form:



The reply is an object of type *QueryDict* which contains all the data returned by the form



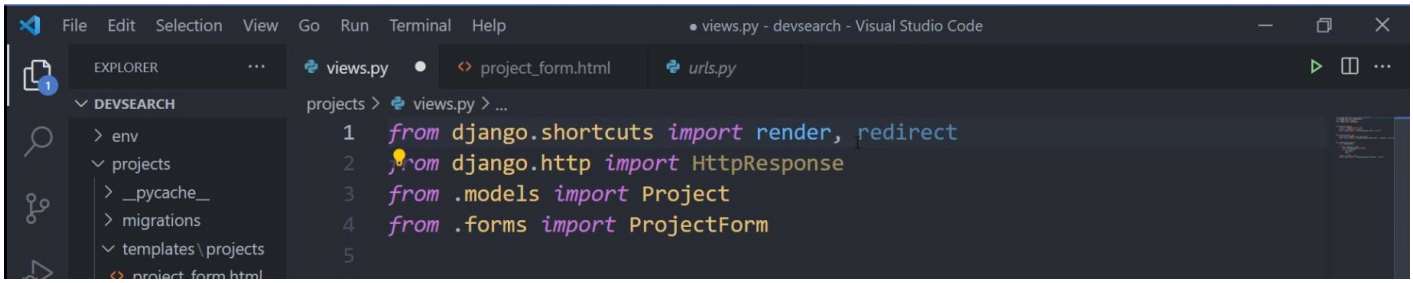
Going back to our /projects/views.py and add the following code:



A beauty of django is its automatic form validation facilities. Django forms auto-check that the contents of each field do not inject code to tables and their data types correspond to each field. Hence, the member function *form.is_valid()* indicate to us that the form complies with all the fields we introduced in our model for that specific form.

If the form is valid, i. e., all the fields pass the default validation tests, we save the form and return to the main page *projects* through a redirection.

The ***django.shortcuts.redirect*** function is a facility that allow us to go to specific parts of the project. We need to include it via an import:



How does *redirect* work?

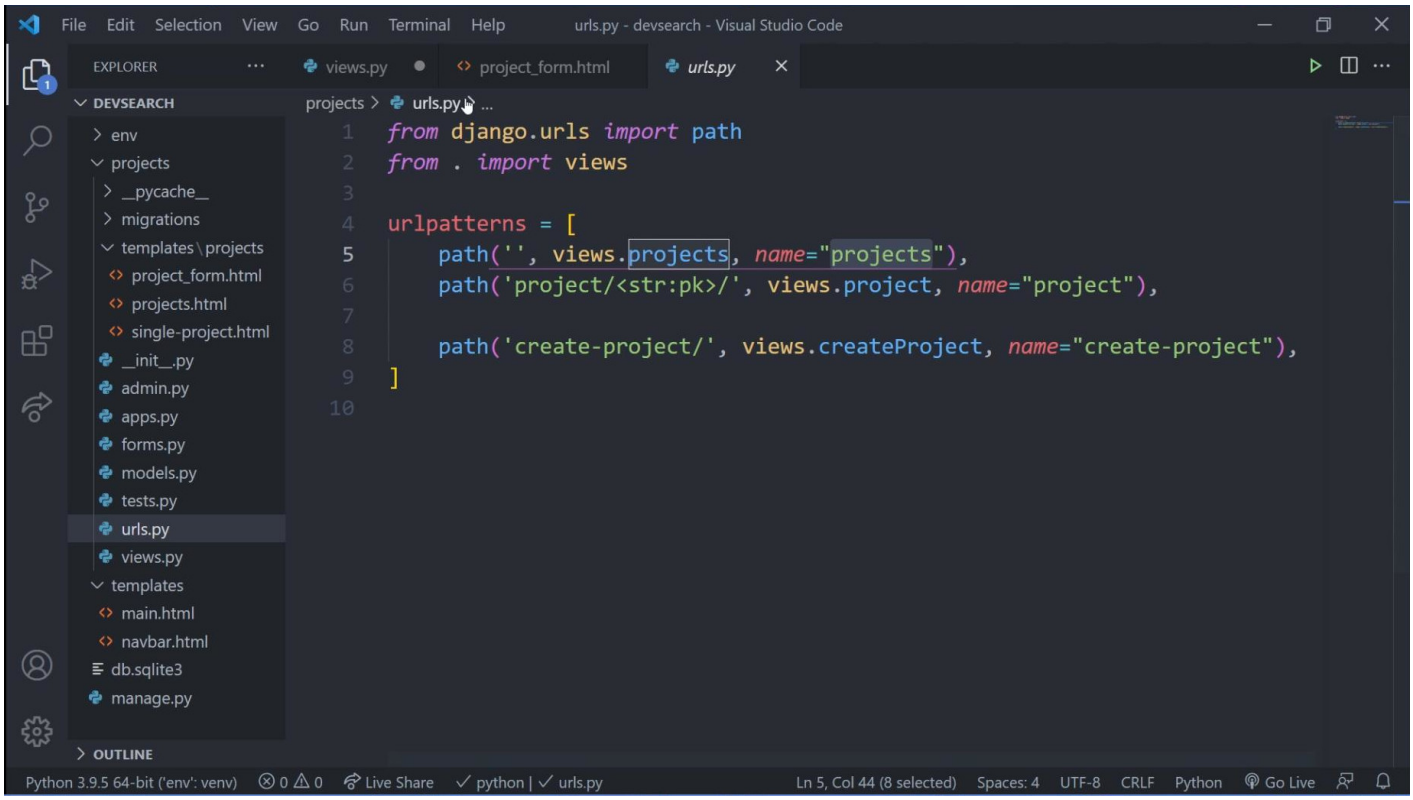
redirect works on dynamic assigment mechanisms we've seen before in our templates with url:

```
{% extends 'main.html' %}
{% block content %}

<p> Hello, you've reach the {{ page }} page </p>

<!-- Let's render the list of projects  -->
<h1> Our Project List: </h1>
<ol>
    {% for project in projects %}
        <li> Project Title: <a href="{% url 'project' project.id %}">{{
project.tile }}</a> --- {{ project.description }} </li>
    {% endfor %}
</ol>

{% endblock content %}
```

Notice that

```
<a href="{% url 'project' project.id %}">{{ project.title}} </a>
```

'project' is the *name = 'project'* in our /projects/urls.py file:



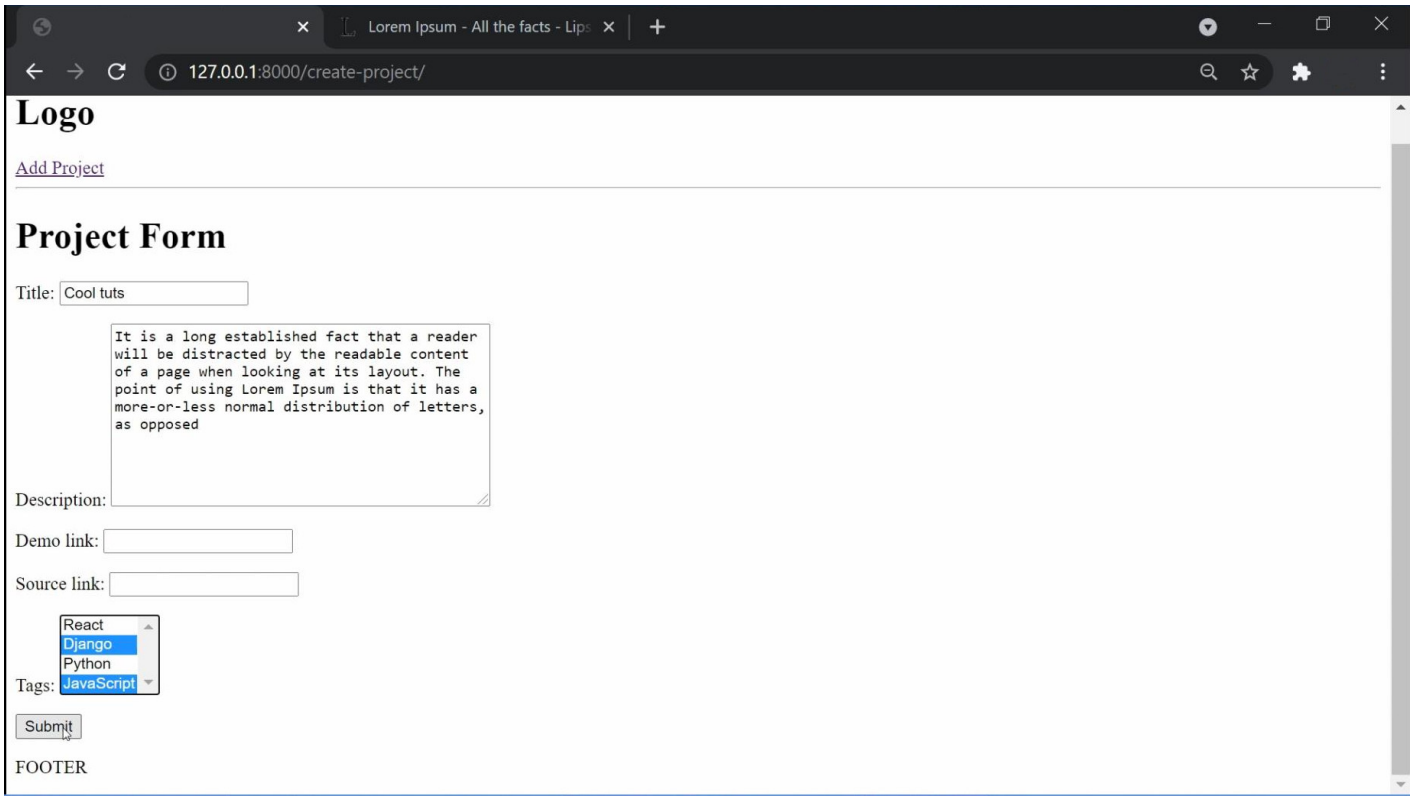
so, for returning to the main page ('projects') page, we use the redirection function

```
if form.is_valid:
    form.save()
    return redirect('projects')
```

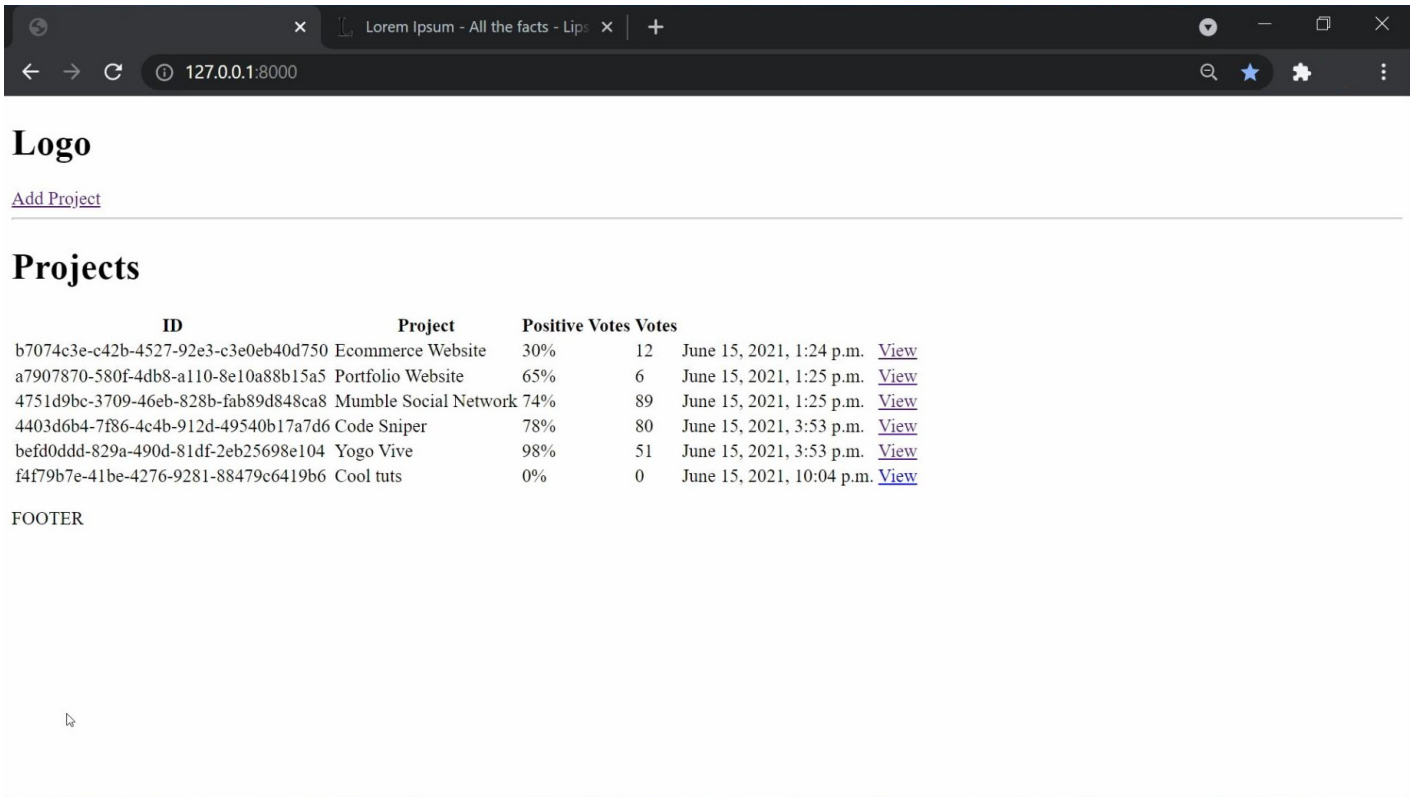
uses the path() with the *name = 'projects'*. The assigment is dynamic.

Insert Data to our Database

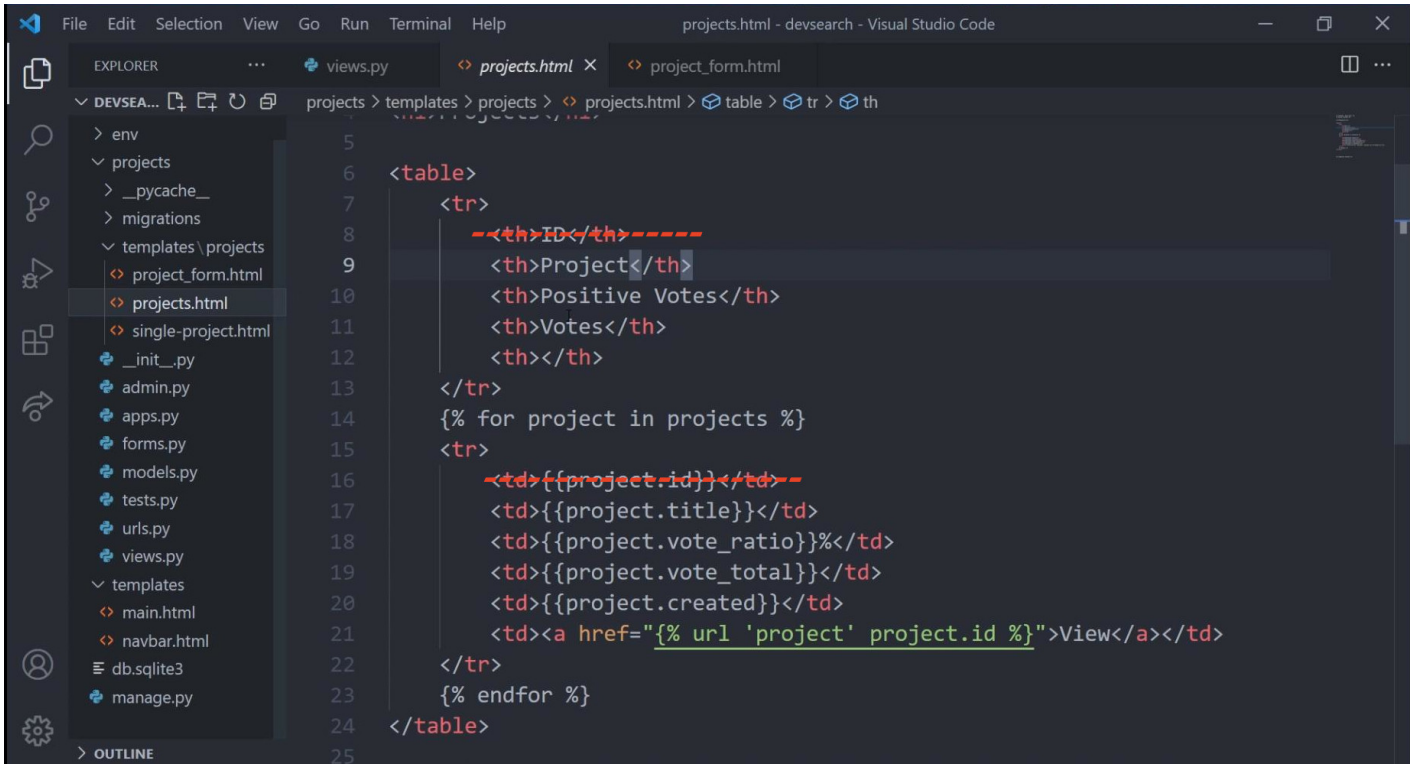
This is our testing phase for the form. Let's introduce some data into our form:



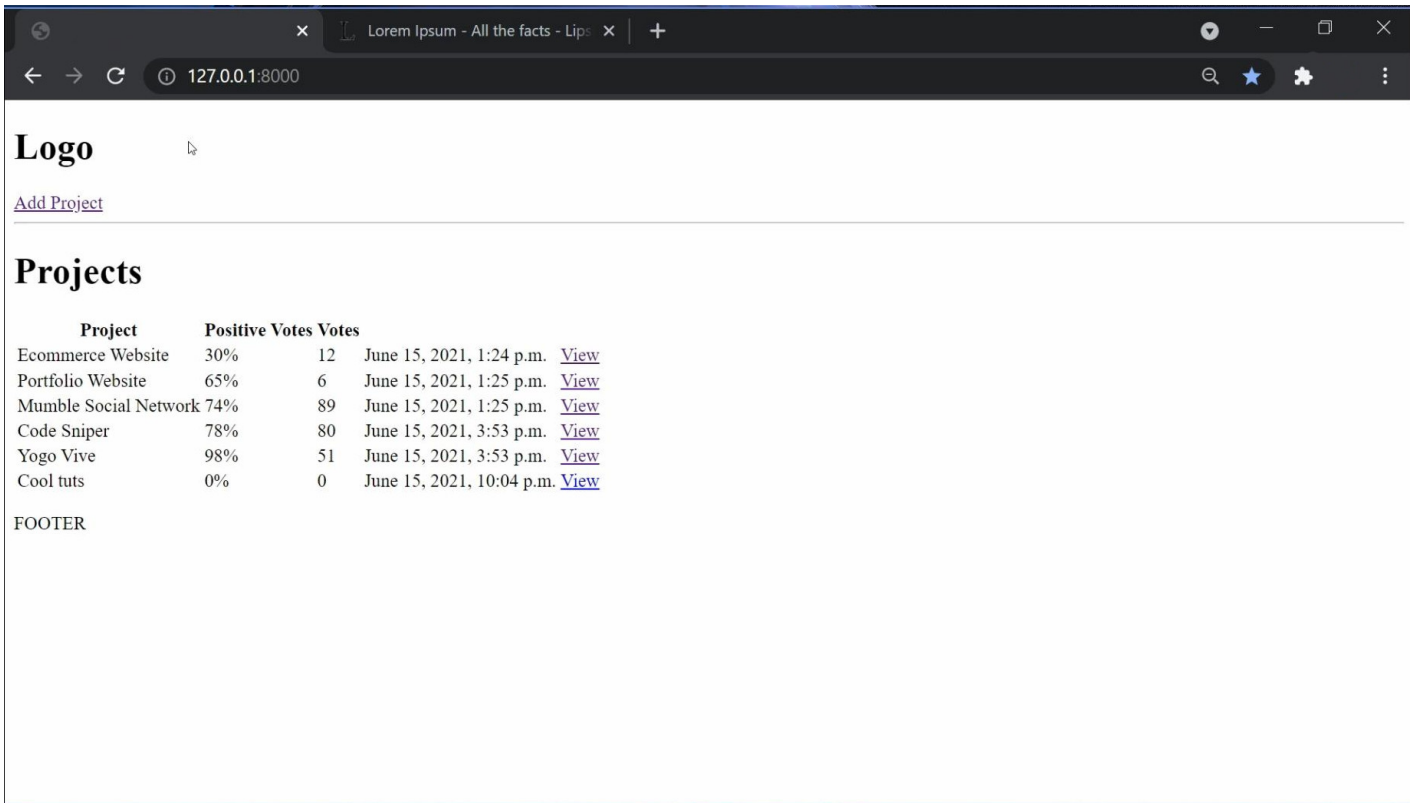
and after submitting the form, you should return to your projects page which should also include the new project:



Let's clean the view a little bit by removing the ID part of the table in the /projects/templates/projects/projects.html:



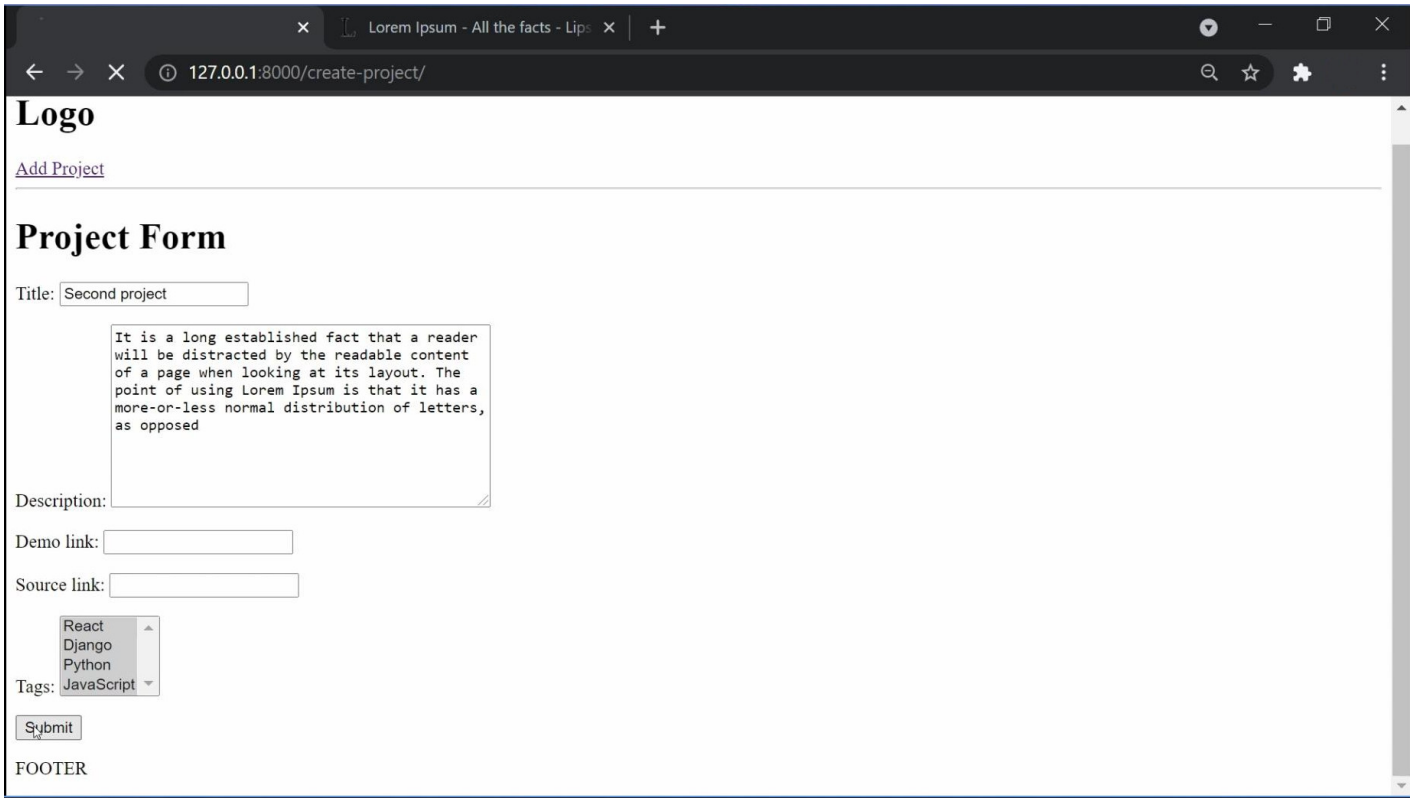
Our template should looks cleaner:



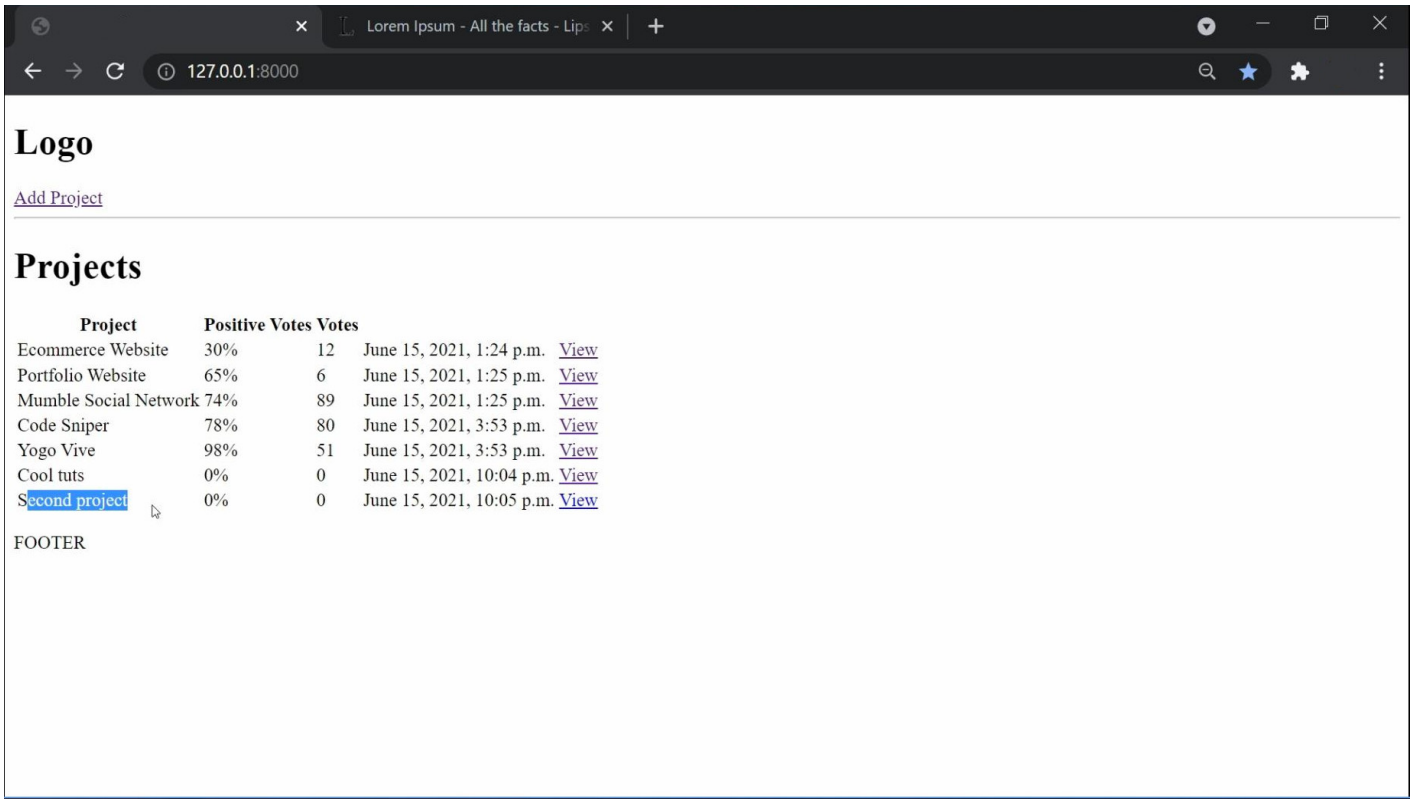
THAT'S IT!

The data is already stored in the database when we used the *form.save()* method.

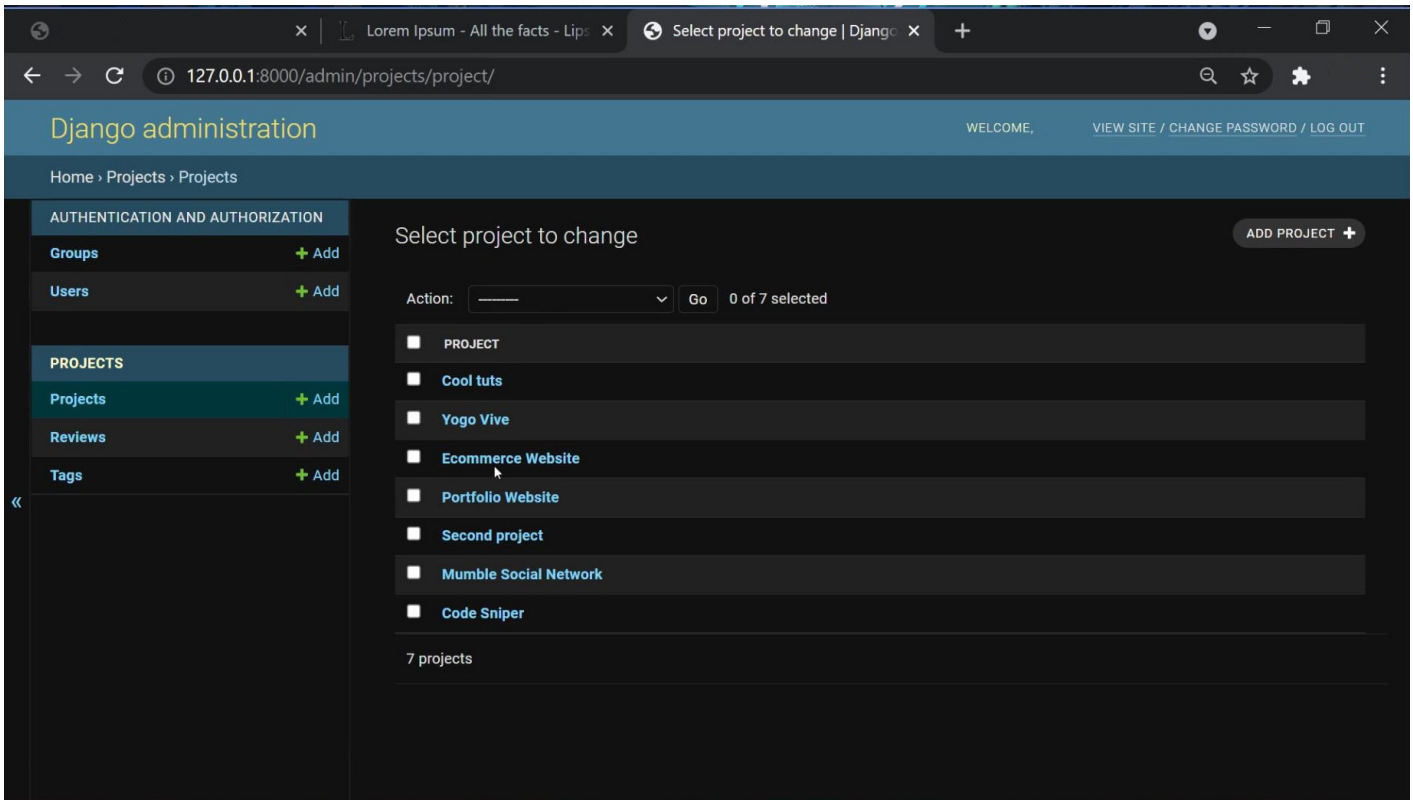
Try more data:



With the new data:



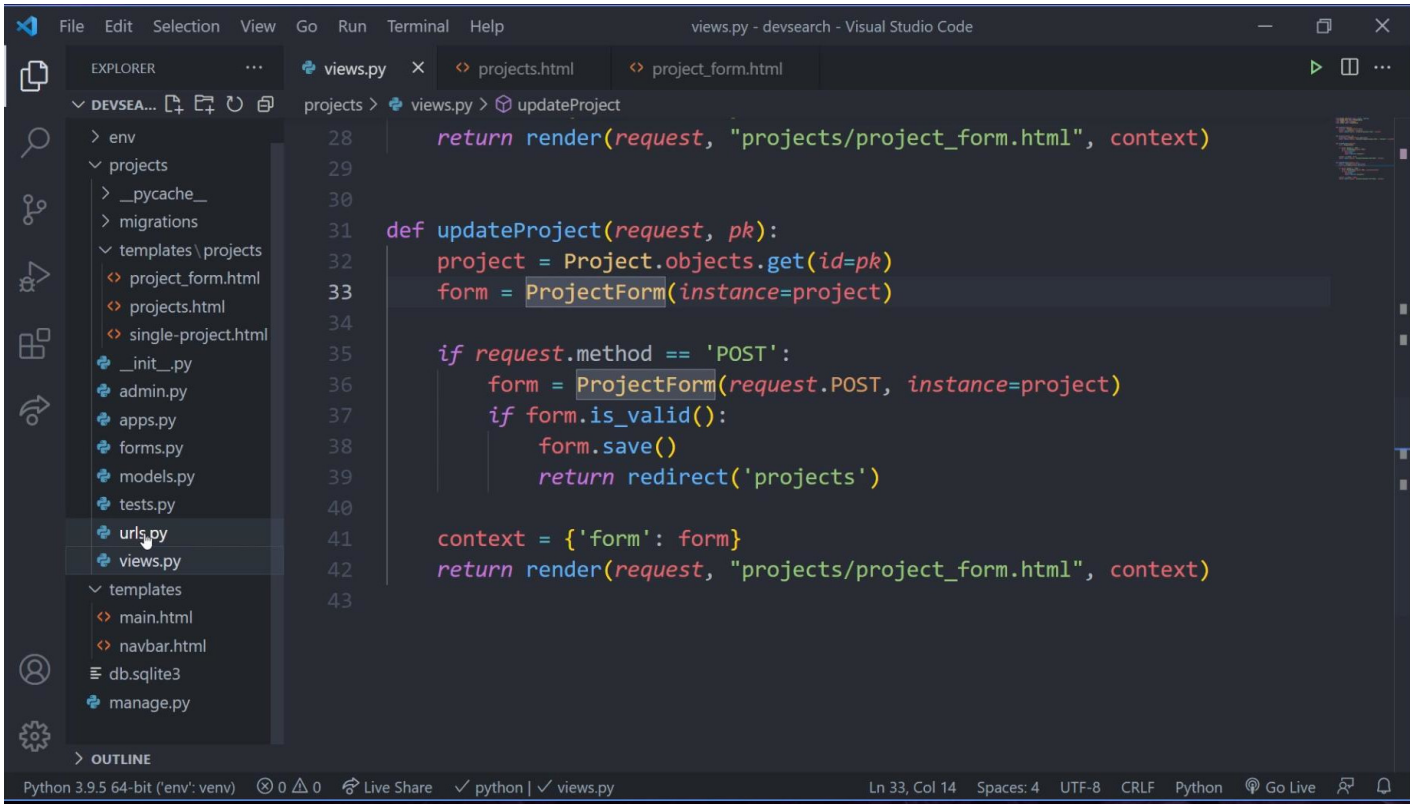
If you enter the admin panel (127.0.0.1:8000/admin/projects/project) you should be able to see the new added projects to the database:



Update Form

In this section we'll create the update form for the Projects application.

The functional class for this form will be very close to the ***createProject*** class, as a matter of fact, we can copy the ***createProject*** class and do some small modifications to the class:



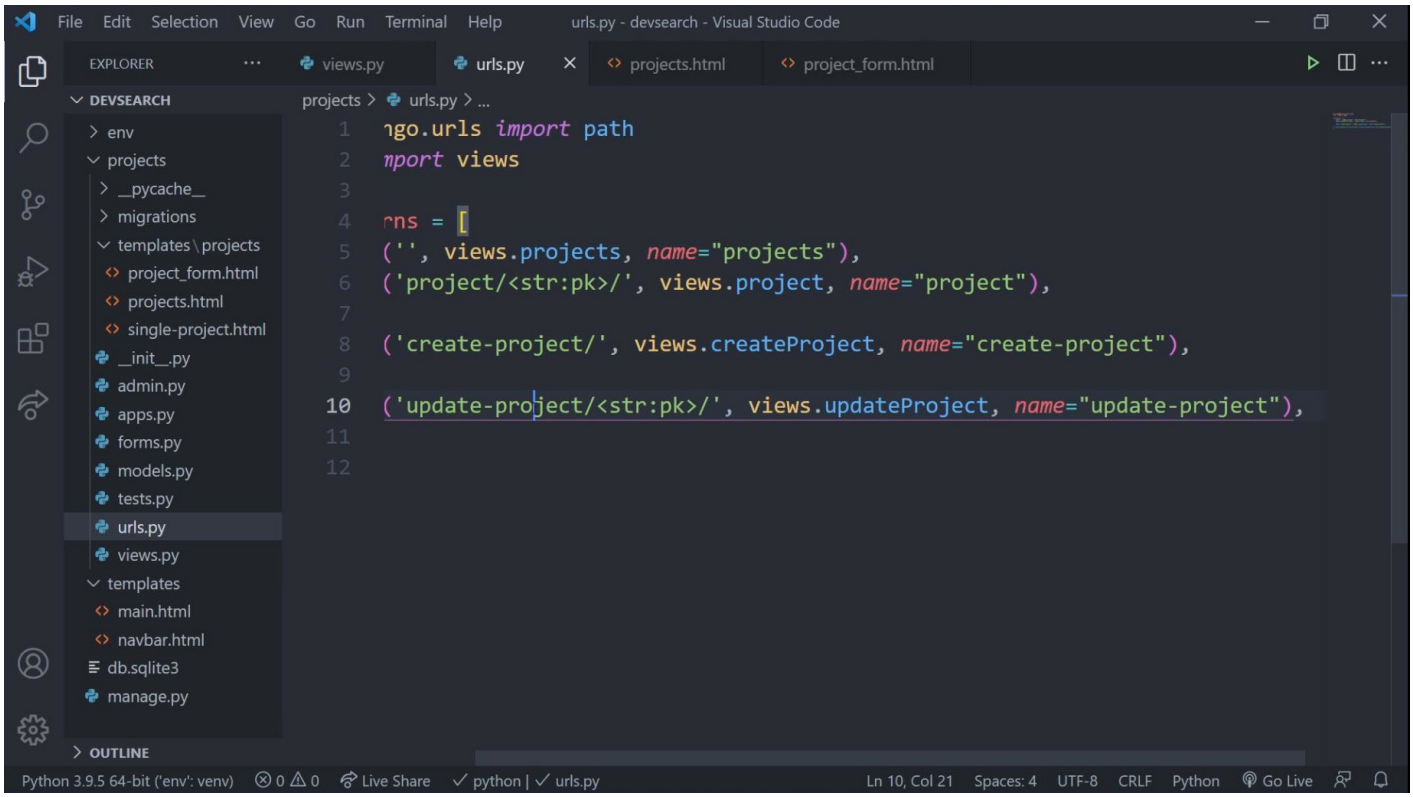
The differences are:

- The extra parameter ***pk***.
- We still need to create the form, but since we're retreiving the data with the ***pk*** information, we need to fill up the form with the same data as in the database, therefore the code:

```
project = Project.objects.get(id=pk)    # Retreives the form data
form = ProjectForm(instance=project)    # Writes the data into the form
```

- We will be implicitly updating the records and thus we'll still use the "POST" HTTP/HTTPS method instead of the "UPDATE" because in this case we're assuming that when the parameter ***pk*** is available, then the data exists in the database and therefore is just an update operation in the database. There's not much difference between "POST" and "UPDATE" except that a design preference. Also "UPDATE" is less commonly used than "POST"
- Since this is an update operation, we'll still be using the same ***project_form.html***

The next step is to update our /projects/urls.py file to include our new ***updateProject*** class.

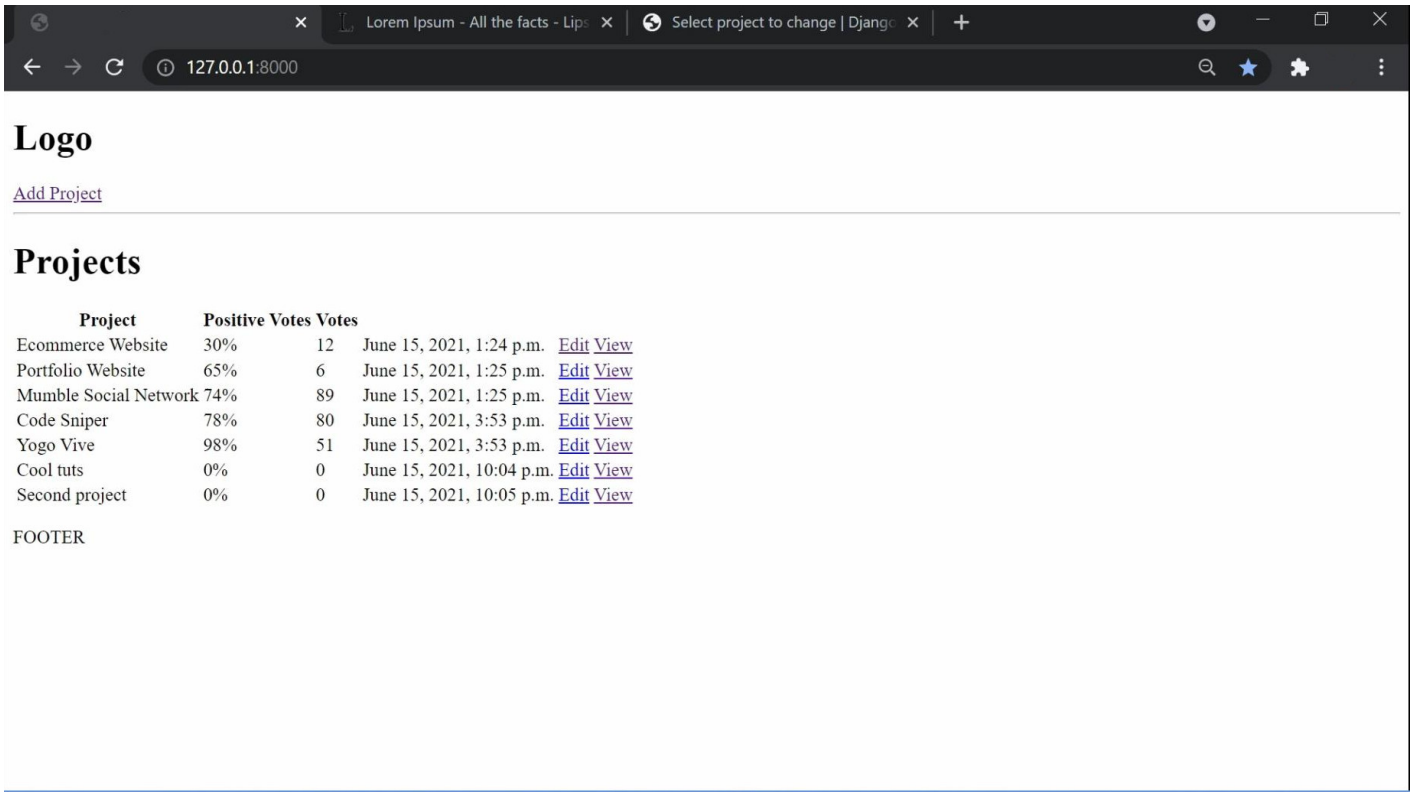


We added the code:

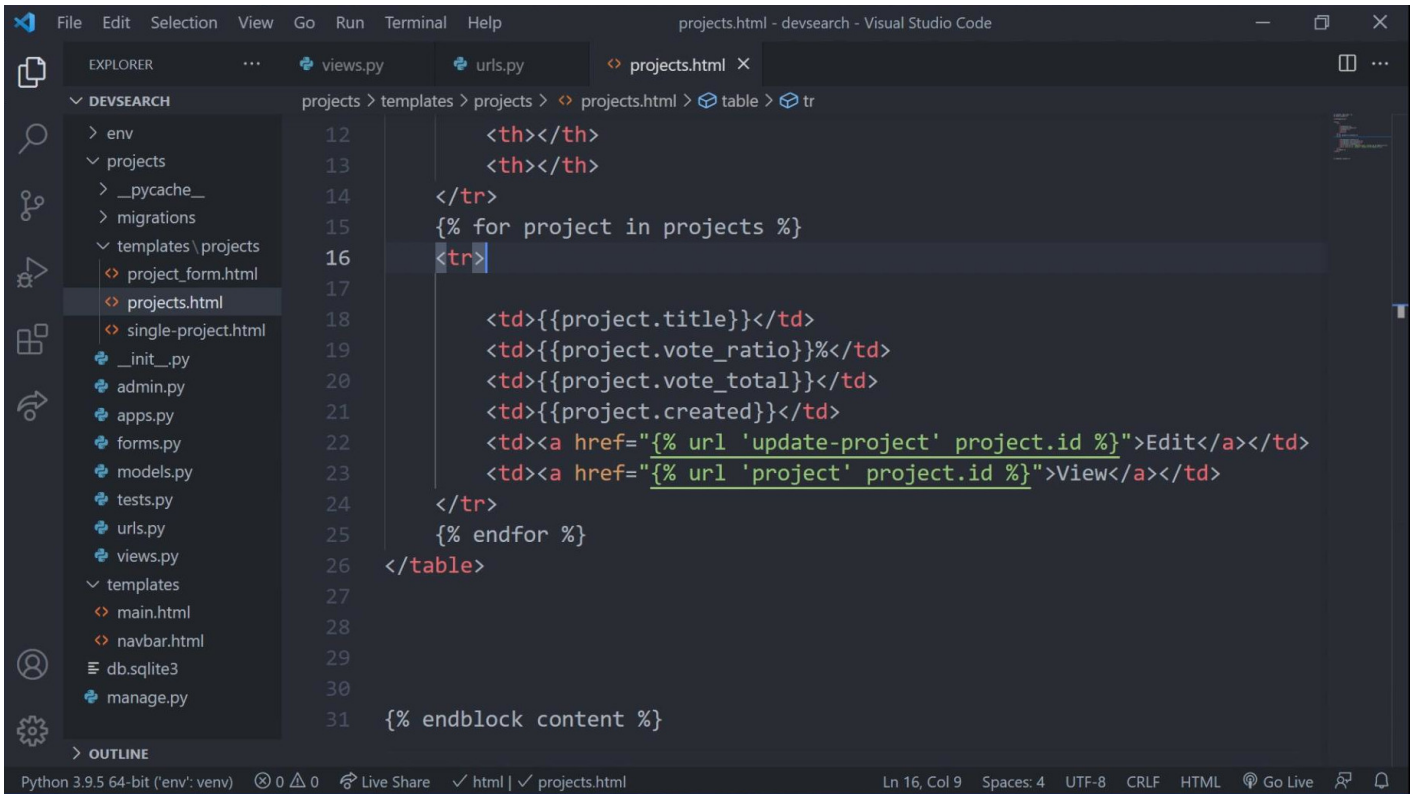
```
    path('update-project/<str:pk>/', views.updateProject, name="update-
project"),
```

to our ***urlpatterns*** list.

Finally, we'll change our projects.html template in /projects/templates/projects/projects.html so we'll be able to see something like the following:



The difference is that we added an hyperlink for form editing. The change in the projects.html is as shown:

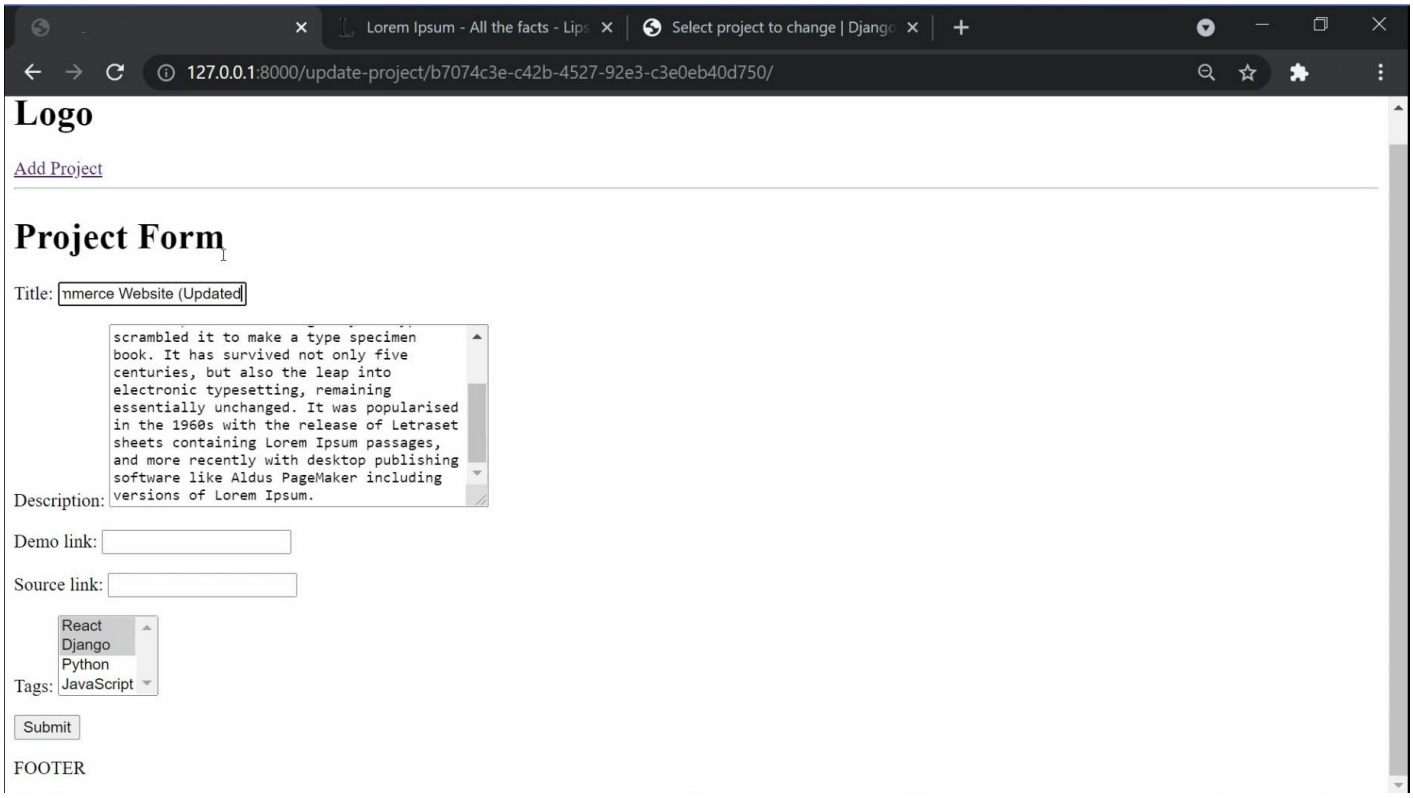


we added the code:

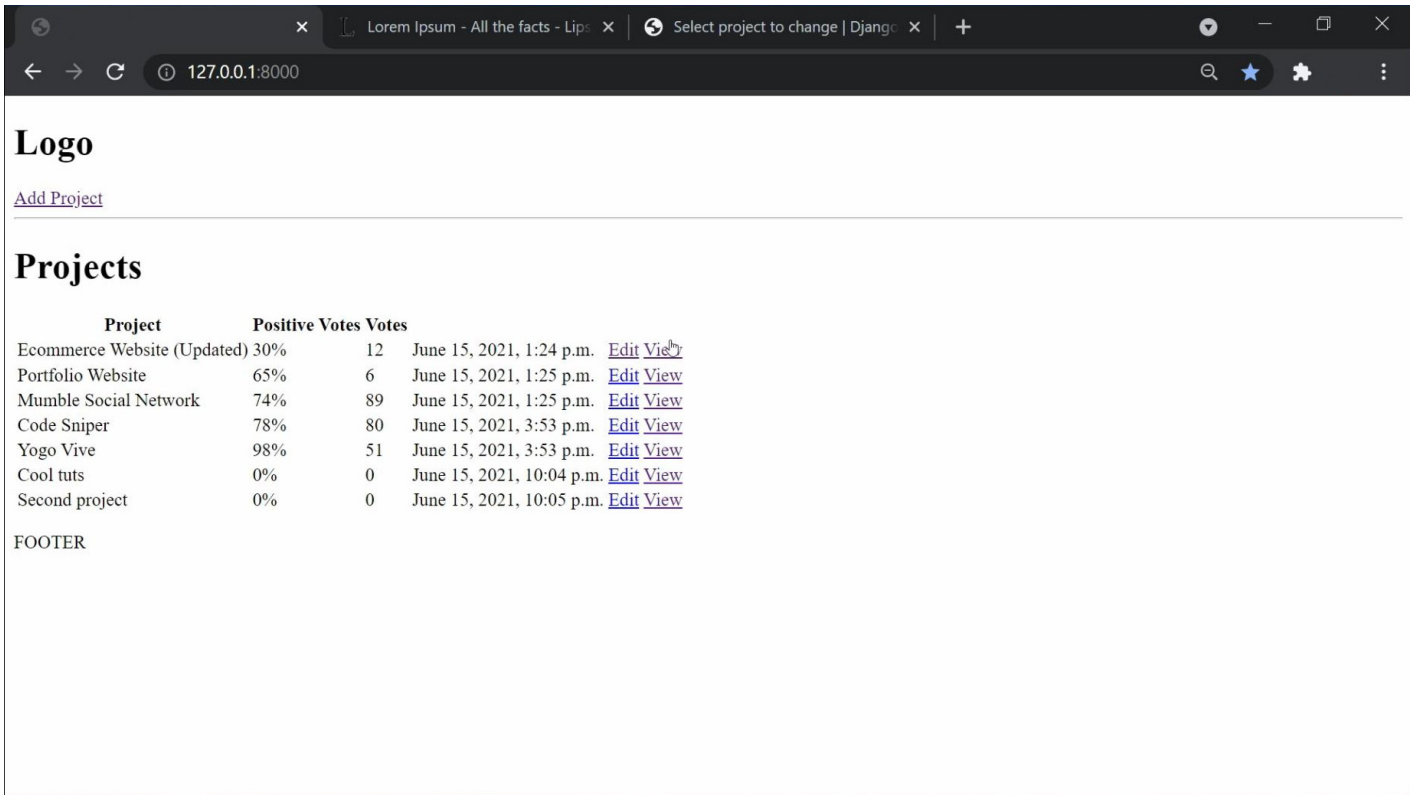
```
<td><a href="{% url 'update-project' project.id %}">Edit</a></td>
```

with in the for cycle.

Test the interface by refreshing the page and click on any project Edit hyperlink, for example:

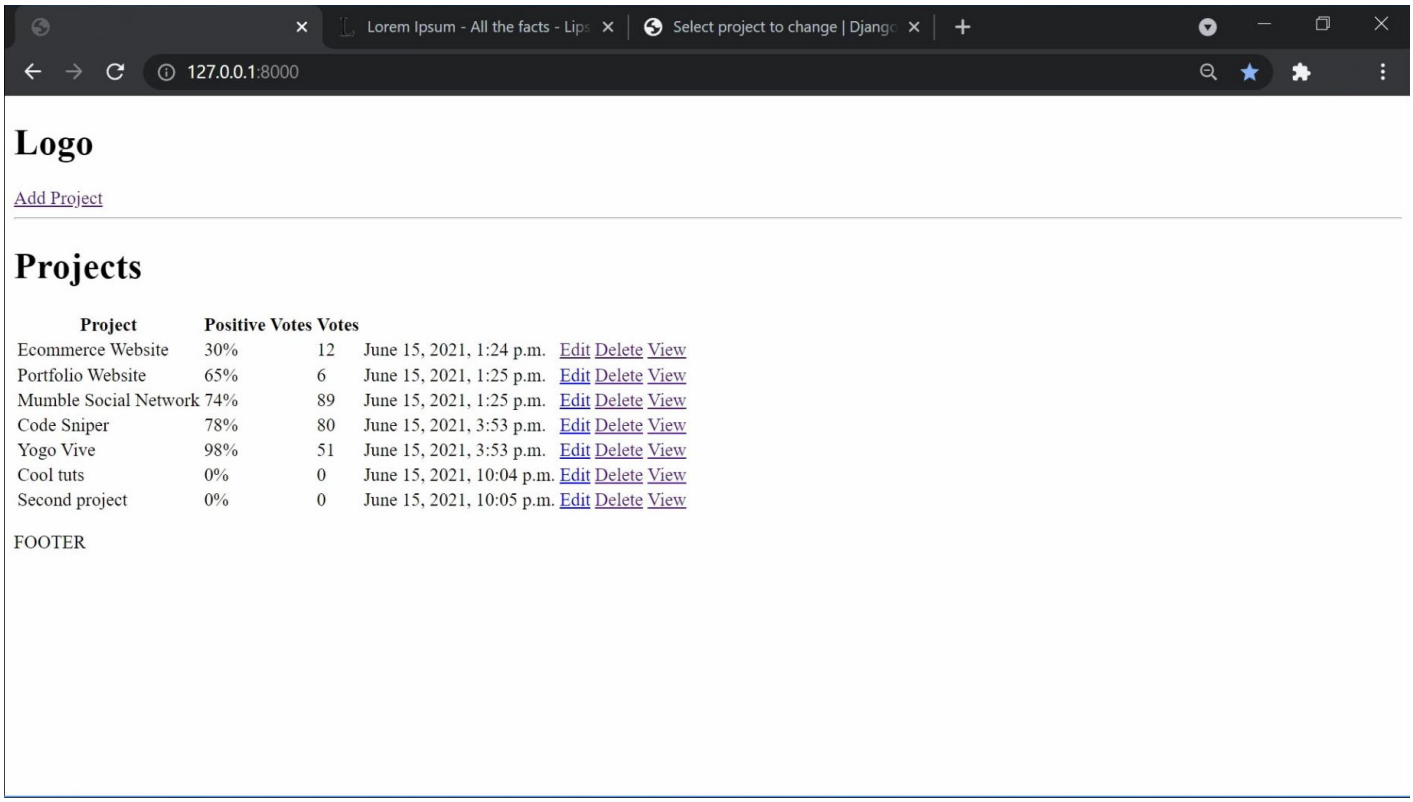


We should see the change in the title with the "(Updated)" text on the project title:



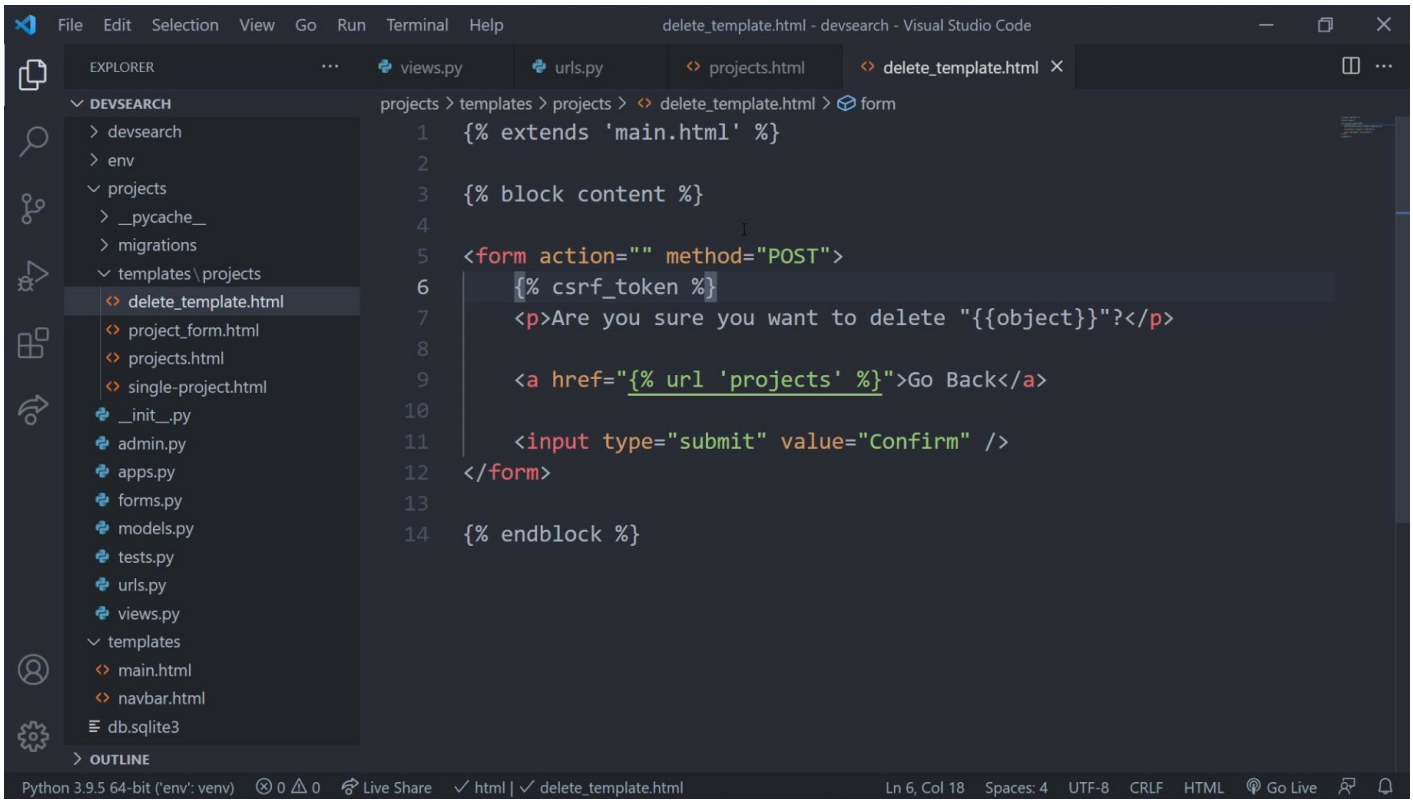
Delete Form

Our objective is now, to modify our projects.html template now to include a Delete operation as shown:



But first, we'll create a confirmation page that shows us the result of the Delete operation instead of simply having the page disapear from the list of projects.

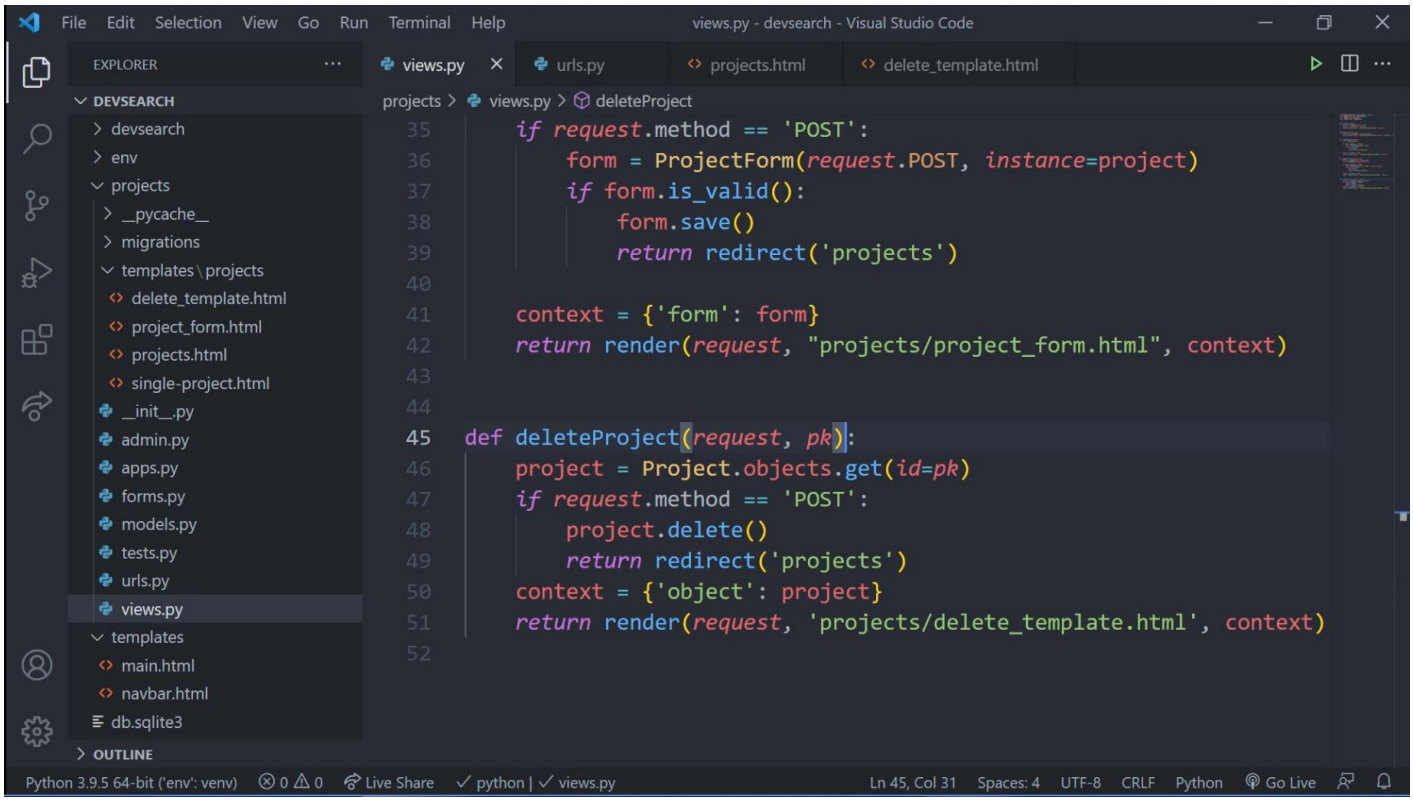
In our /projects/templates/projects folder, we now create a new template delete_template.html:



Few things to notice:

- The ***csfr_token*** that is always present in any template.
- Now our template parameter is "***object***." This makes the template generic enough for other pages since is not specific to ***projects***.
- The template is only for confirmation and thererfore uses a botton for such confirmation.
- To cancel the ***DELETE*** operation, we simply return to the ***projects*** page.

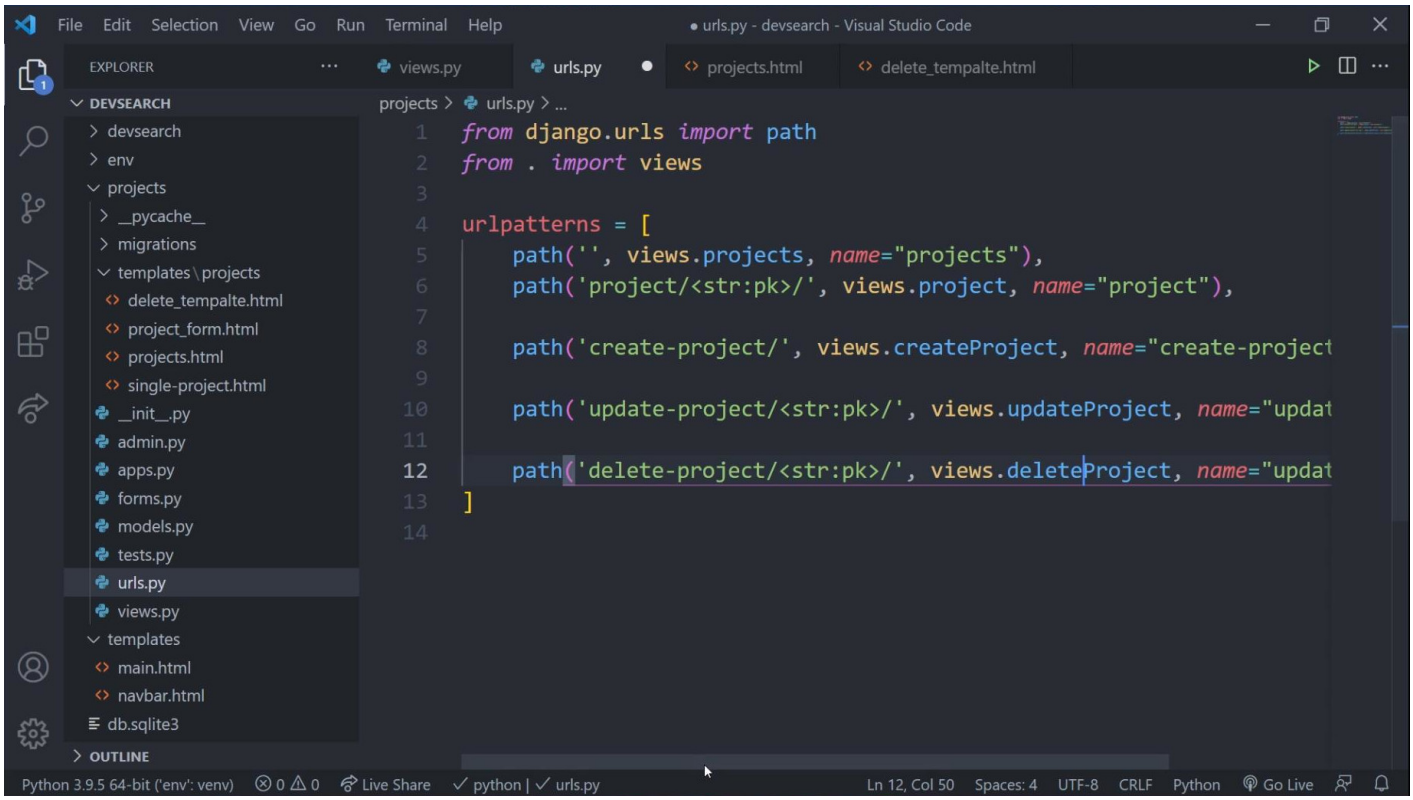
Our next step is to create a routing class for the delete operattion in /projects/views.py:



Notice three important things:

- Again, we're using the "POST" method, the reason being that it's easier to intercept when dealing with interactive UI. If we were to use "DELETE", we would had had to generate the method manually after clicking the submit button.
- We are using the paramter ***pk*** again to specify which record has to be deleted from the database.
- The argument ***object*** is used to pass the data to the confirmation template (***delete_template.html***)

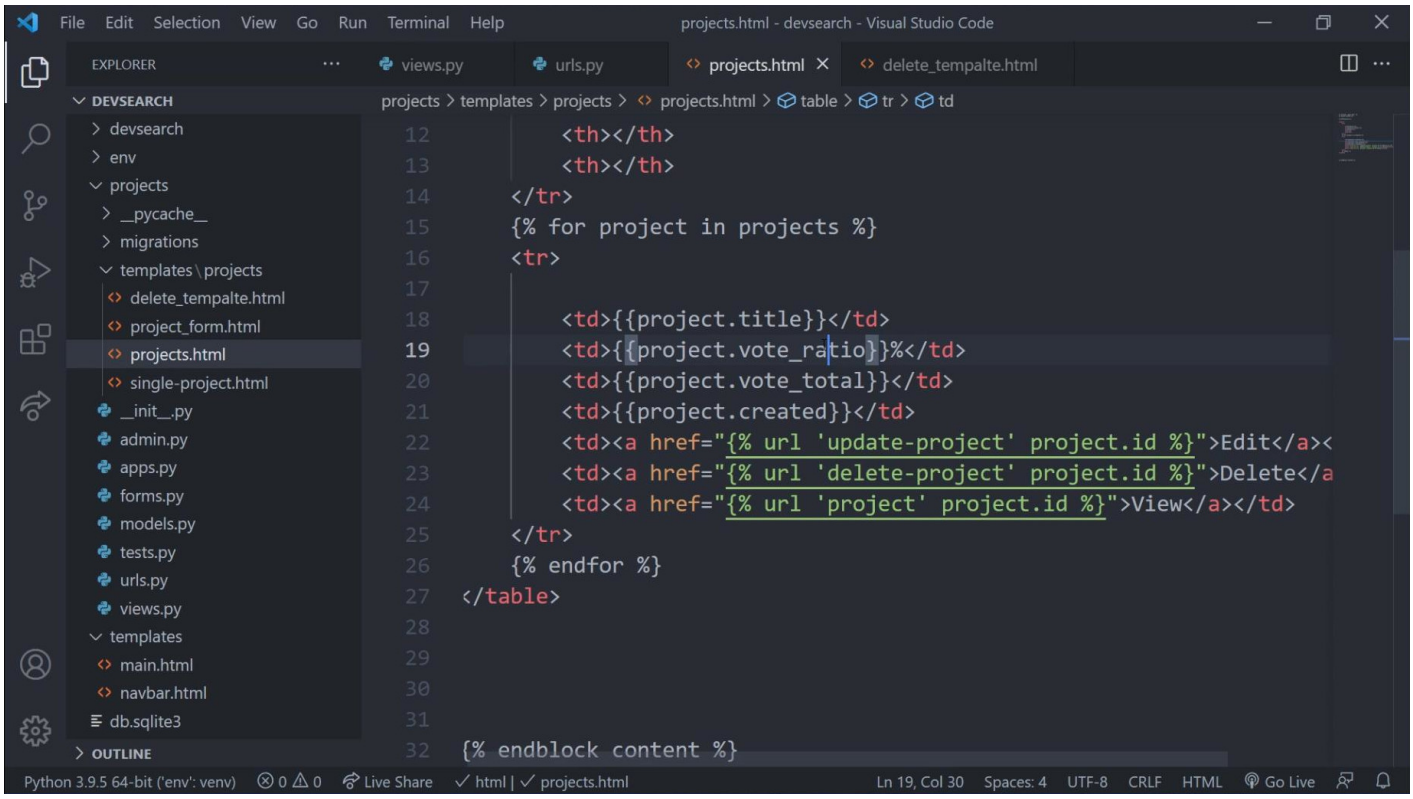
Still, we need to route the routing class by adding its path in the /projects/urls.py file:



We added the code:

```
path('delete-project/<str:pk>', views.deleteProject, name="delete-project")
```

Now we'll be back to our projects template (projects.html) to add the necessary code to include the deletion view:

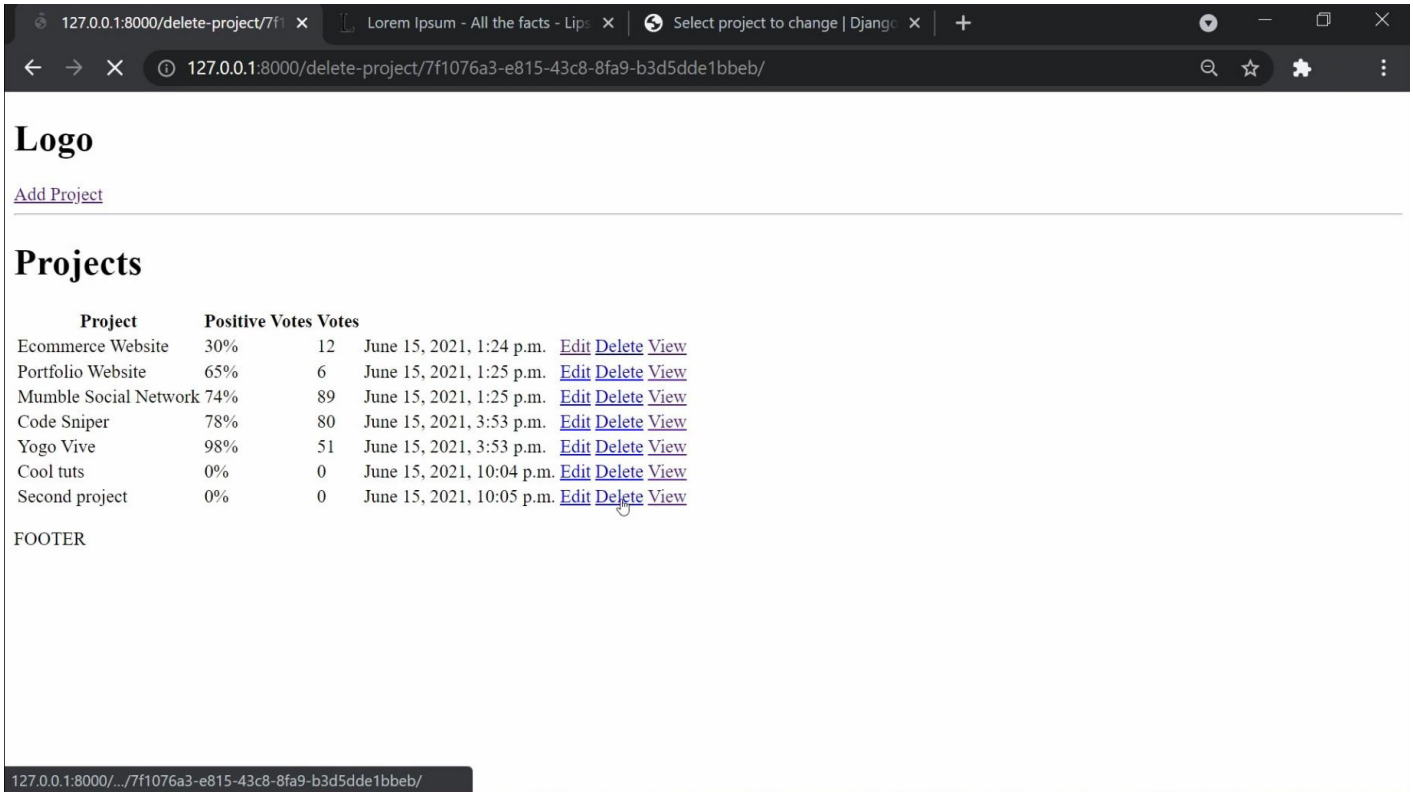


where we add the code:

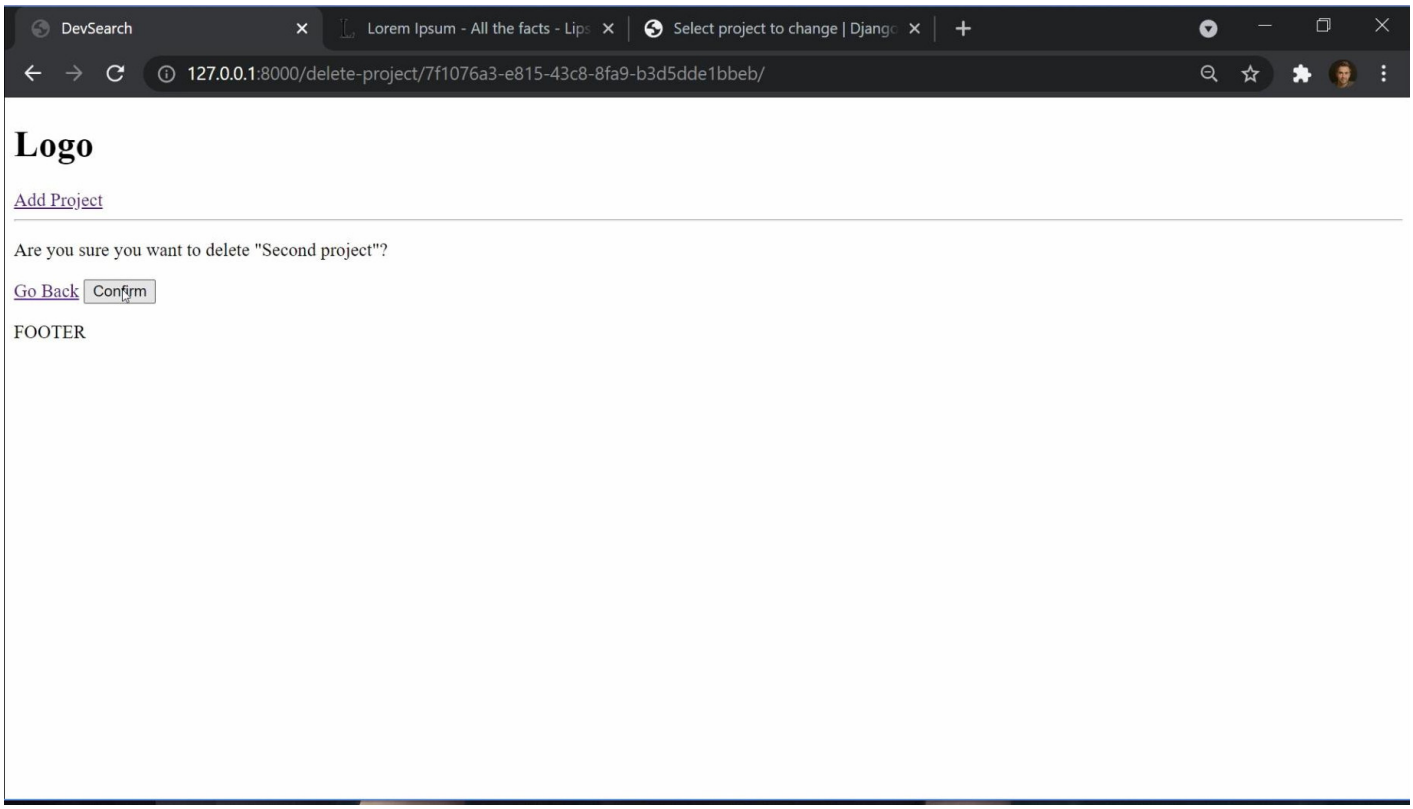
```
<td><a href="{% url 'delete-project' project.id %}">Delete</a></td>
```

This should be the last step.

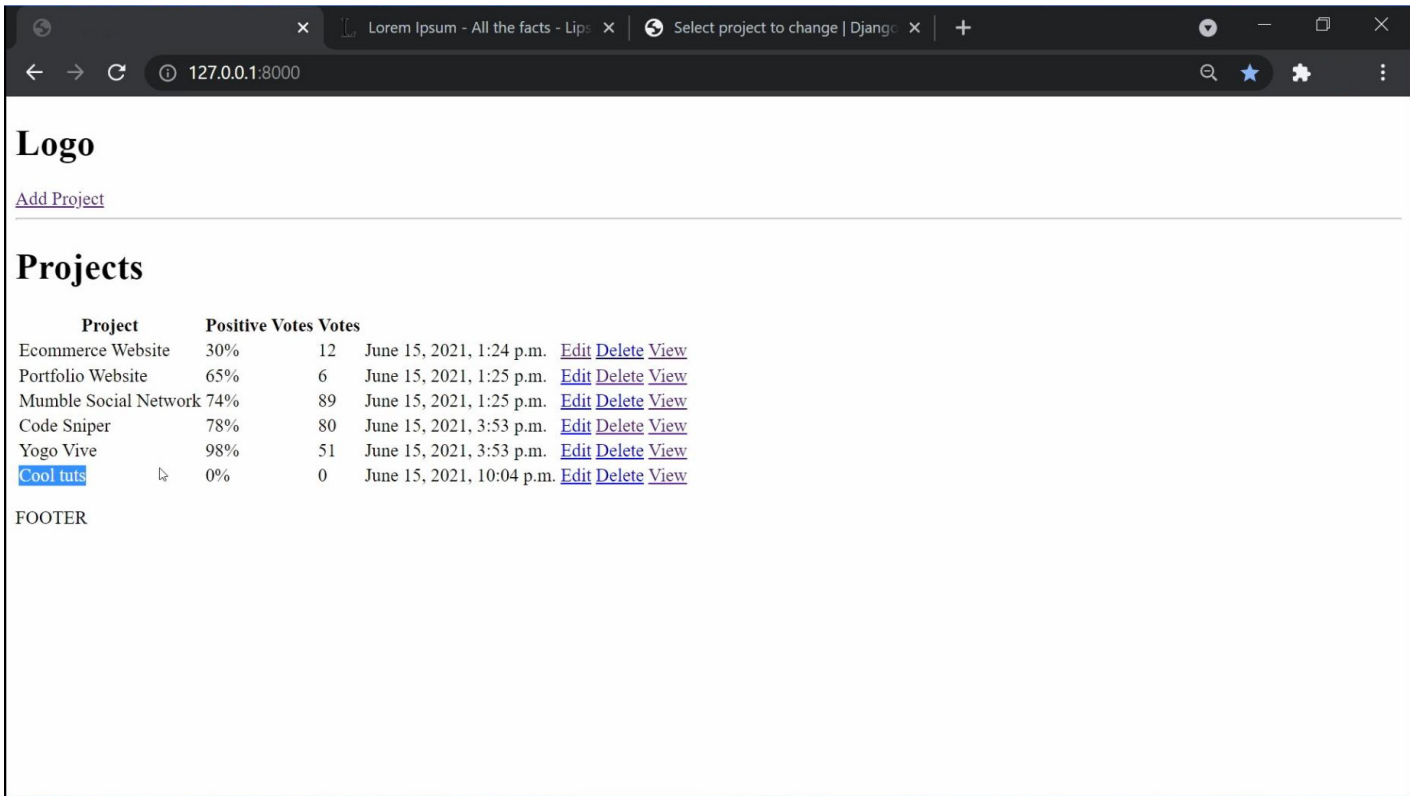
Let's test the deletion process. Open up the projects page:



and click on any project, in this case, it's the last project. Then after clicking on the "Delete" hyperlink, you should be able to see the delete confirmation template for the specific project you had choosen:



The projects page now should look like:



Test Your CRUD

Using the newly created forms, delete a couple of projects, create and edit projects adding some kind of data that shows you that they are actually working on the database.

Conclusion

With this, we've seen the basic forms that do basic CRUD operations on our database. We checked the files and templates necessary to implement such operations in our site.