**Building Dynamic Webpages**

**Create a dynamic template**

Now that we have our notes, how about we learn *how to display them*?

Let’s create a view.

* Go to note -> views.py

And let's start with importing the models - Note.

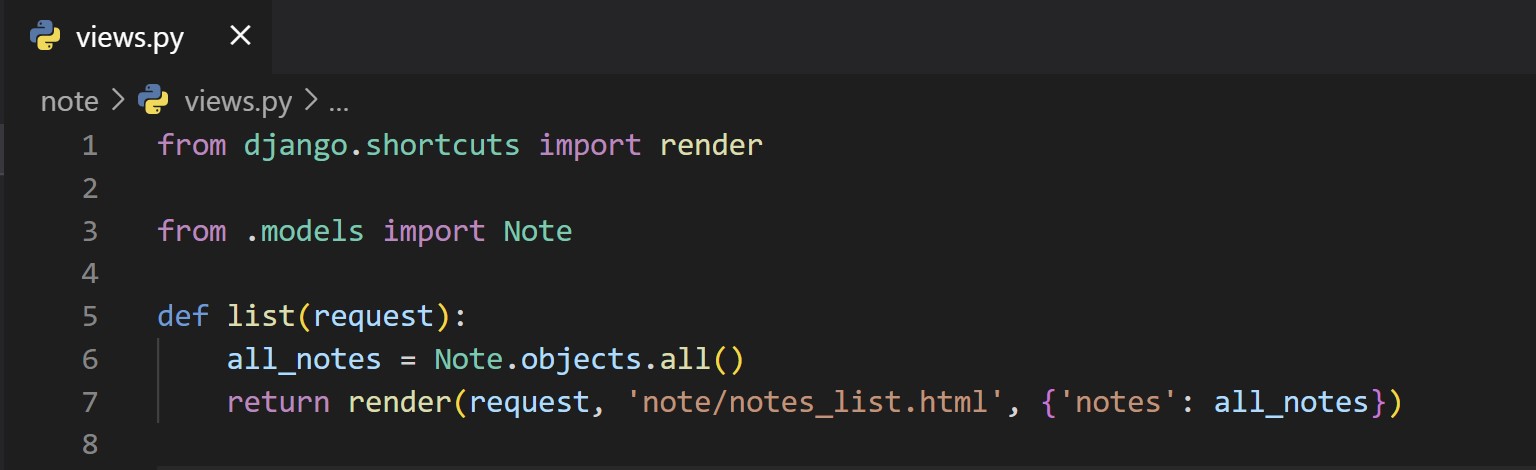
* Type -> from .models import Note

Now let's create a function called ‘list’ that receives a request, and then a variable all\_notes that stores all the notes that we have in our database. Return the render function with ‘request’, a template that we're going to create a little bit later  ‘notes/notes\_list.html’ and in the brackets with ‘notes’ are equal to ‘all\_notes’.

* Type -> def list(request):

all\_notes = Note.objects.all()

return render(request, 'note/notes\_list.html', {'notes': all\_notes})



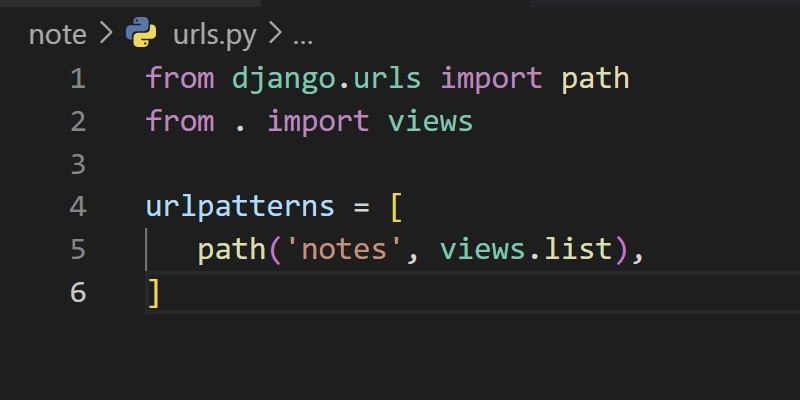
Here are querying for all notes and sending them to the template. This way, when the template is rendered, all the information coming directly from the database will be available.

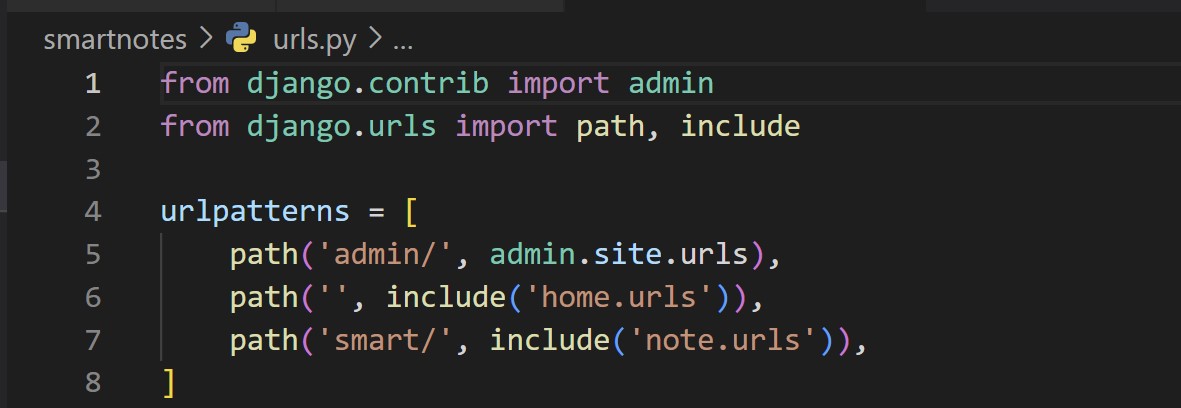
Before we go to the template, let's organize URLs.

So let's create a new ‘urls.py’ file in ‘note’. And that's going to have the same format.

Type -> from django.urls import path

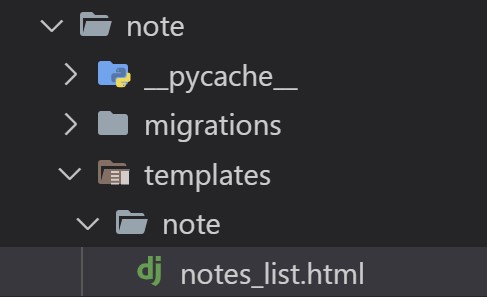
Then let's import the views here. And then add the urlpatterns that has a list. See image below.



The last thing is that we have to add this on the urls.py file on smartnotes. See image below.

By doing this, all the URLs that we are adding on ‘note.urls’ will be added after the ‘smart’. So ‘smart’ is going to be a part of that endpoint. This is a really nice way of organizing the project.

Now we need to create the ‘templates’ folder in ‘note’. Then a new folder ‘note’ in ‘templates’, where we add our template ‘notes\_list.html’.



 Now we can create our template. So, type ->

<html>

<h1>These are the notes</h1>

<ul>

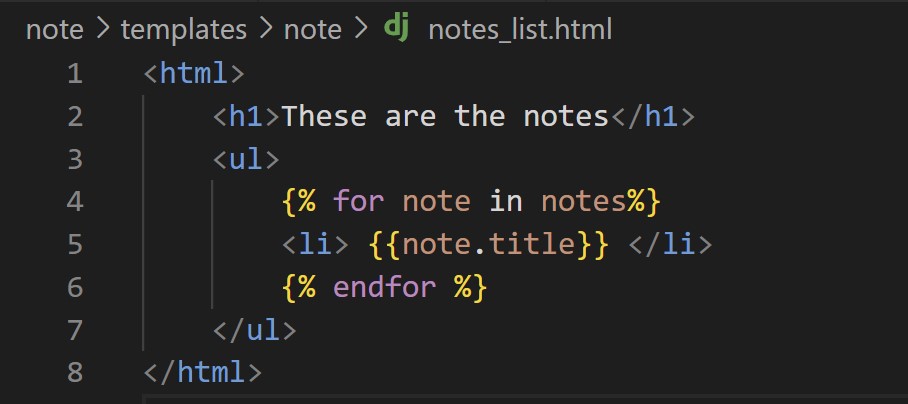
{% for note in notes %}

<li> {{note.title}} </li>

{% endfor %}

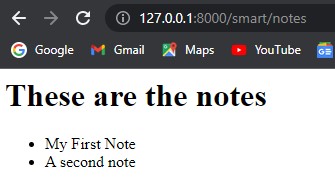
</ul>

</html>



Everything that is between curly brackets is the Django template language logic. Here we're opening a list tag ul, and then saying that for each note we receive in the template, DTL should create a list item, the li. Notice that commands, such as the loop happen between curly brackets and percentages, while things that should be rendered by the template are between double brackets.

So let's save this. Then run the server. Open ‘http://127.0.0.1:8000/smart/notes’. And here are the notes!



So, the web page is dynamically getting data from the database and adding it to the HTML. Now, you can go and create more notes, either via the shell or the admin and see what happens here.

**Display content of a single note**

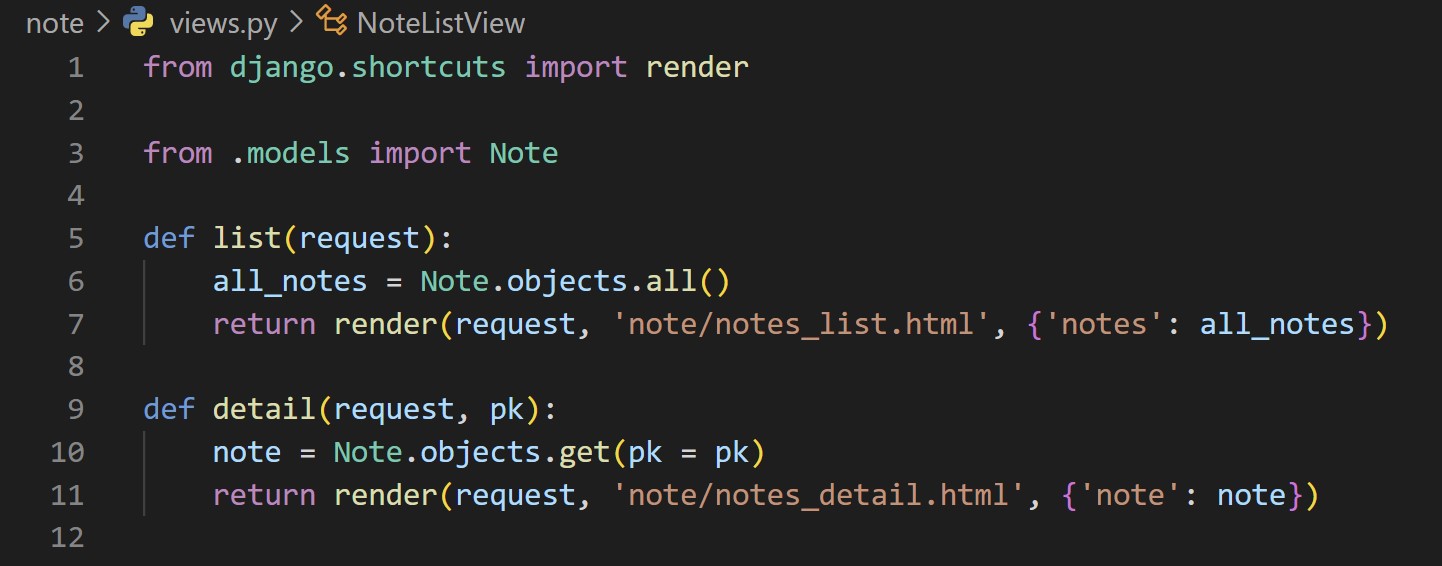
Now that we have a list of notes, how about we create a way to visualize details of a particular note?

Let's go back to the notes app, views.py and let's create a new function here. Now, this function should receive a second parameter called pk for private key so that we can go in the database and get that particular note. So, let’s write:

def detail(request, pk):

note = Note.objects.get(pk = pk)

return render(request, 'note/notes list.html', {'note': note})



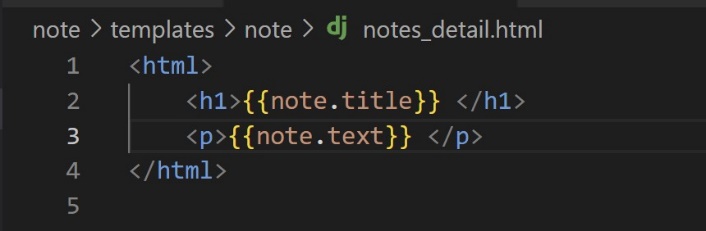
So, now what we need to create the template called ‘notes\_detail.html’ the same way as we created ‘notes\_list.html’. And let's create a simple HTML:

<html>

<h1>{{note.title}} </h1>

<p>{{note.text}} </p>

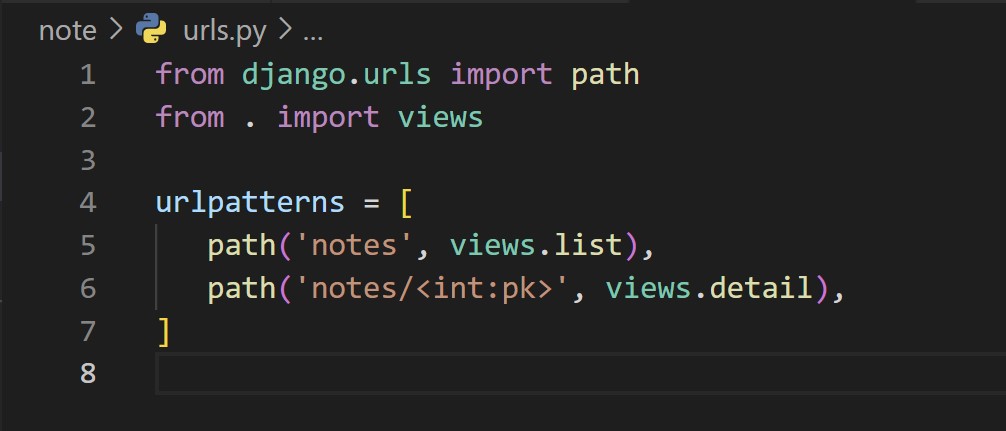
</html>



There you go!

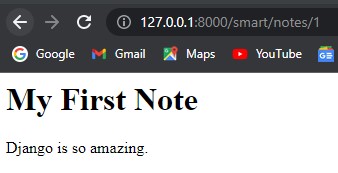
There's one thing still missing, which is the URL. This needs to be a slightly different URL because we need to be able to pass down the second parameter to that function. So let's do this by adding a new path in ‘urlpatters’ in ‘urls.py’.

Add -> path('notes/<int:pk>', views.detail),



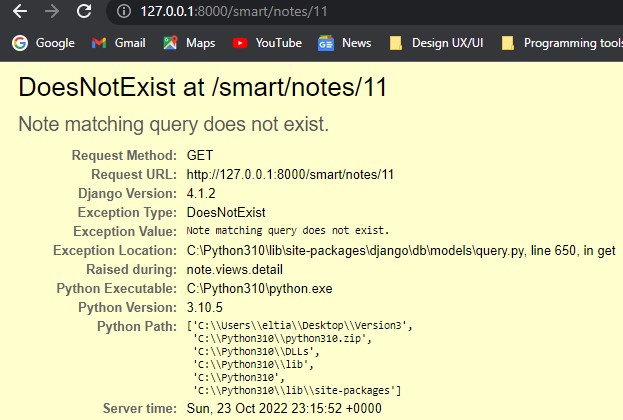
We are telling that URL will receive a new value named pk that will be an integer number. Now, the only thing left to do is start the runserver again. And test this out.

So if we pass now the pk for our first note, we can see the template displaying the details of the first note.



So this works fine but we still have a problem. The get method that we're using to get the note from the database will actually throw an error if you pass down a private key that doesn't exist. So if we try the same URL but for example 11, unless you have created 11 notes, this will raise an error.

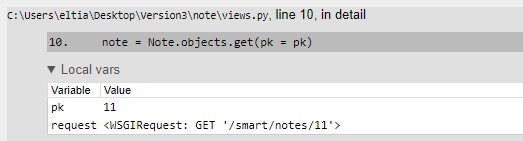
So let's try it out.



Notice here that this is returning an exception of the type ‘DoesNotExist’. We can also see here that there is a message with the exception saying ‘Note matching query does not exist’.

Django also has an amazing traceback for us to understand where exactly the error happened.

You can see right here that the problem started in line 10 on the notes, views.py file, which is exactly where we define the query.



We only have this page explaining the error because we continue to have the debug equals to true in the settings file. In a production environment, the user would see a 500 error, which means an internal error. When an object is not found, the correct response is a 404 status code saying that that object does not exist. So let's change our code to make sure that we get the correct status code.

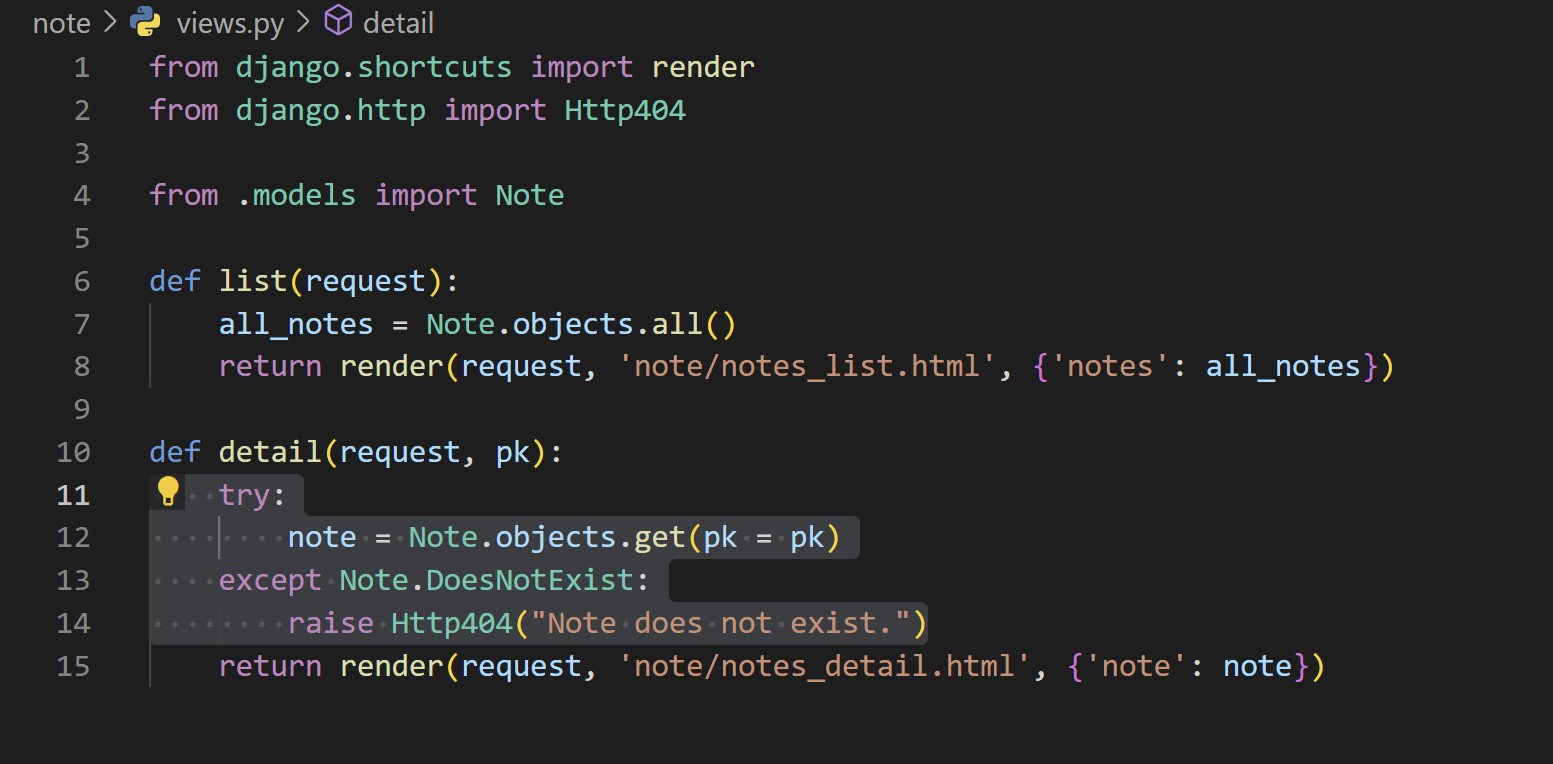
Let's go back to the views.py file in ‘note’. And in here, let's import -> from django.http import Http404. Now we can wrap the query in a try and except block ->

  try:

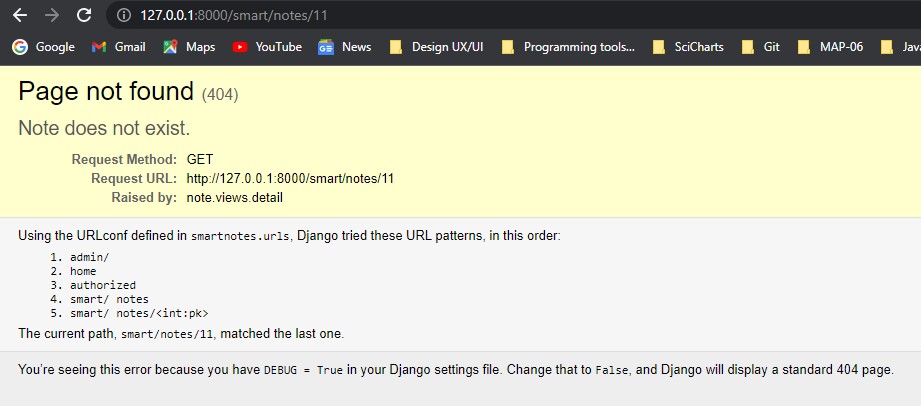
note = Note.objects.get(pk = pk)

except Note.DoesNotExist:

raise Http404("Note does not exist.")



If you go back now to the previous link, and refresh, we're going to see here that this page is now returning a 404 with a message that we define. This is a much nicer flow than the error we had before because we're controlling the message to the user.



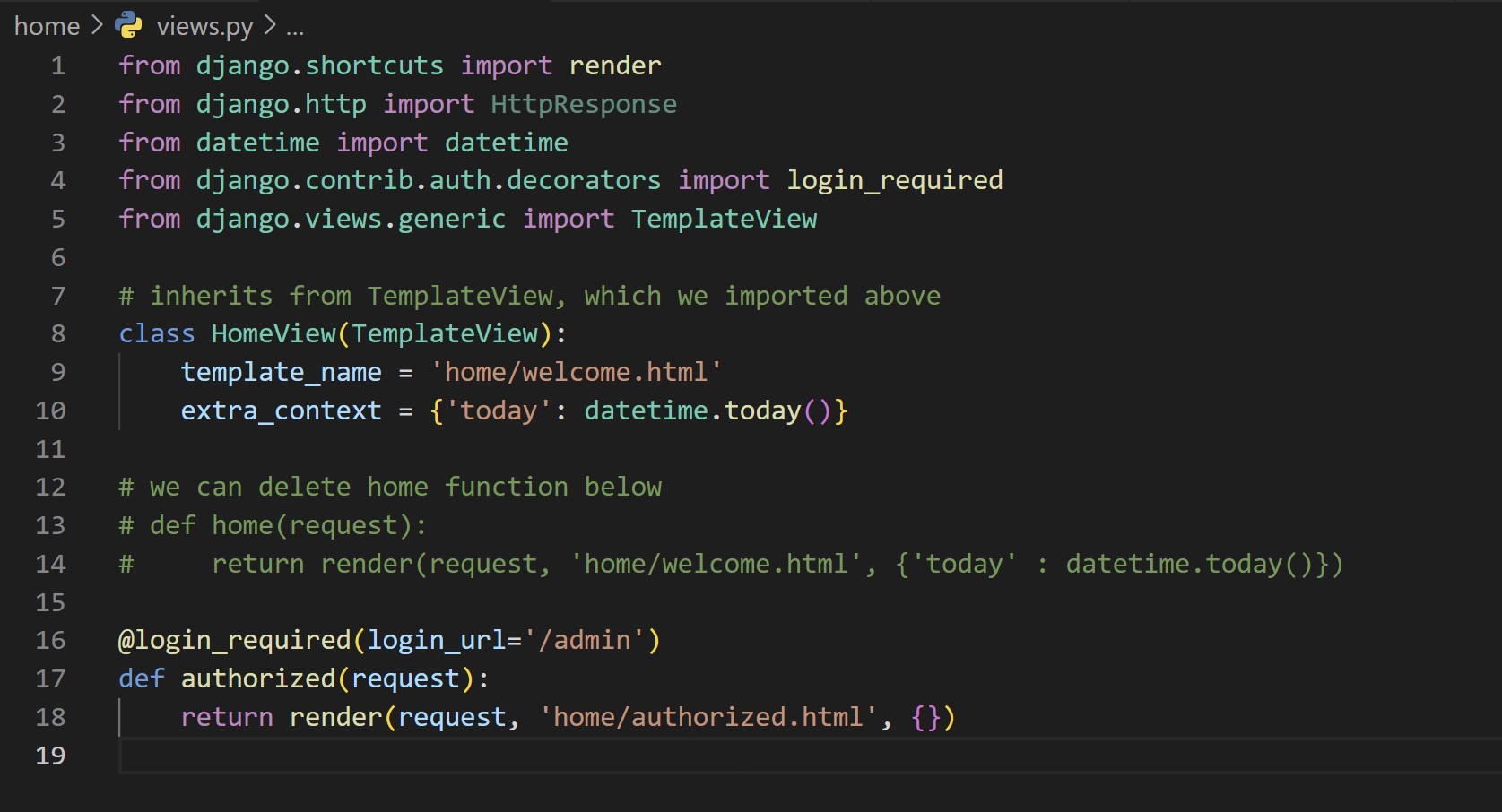
**Introduction to Django class-based views**

So far, you've learned how to create views using functions. However, Django has a couple more features that we can leverage to get things even simpler. Welcome to class-based views! Most views have similar patterns, and reinventing the wheel is something nobody really wants. Class-based views are extensive classes that implement typical view behavior, and you just need to override a few things to make it do what you want. This will allow us to avoid the boring work and focus on the things that are unique to our project.

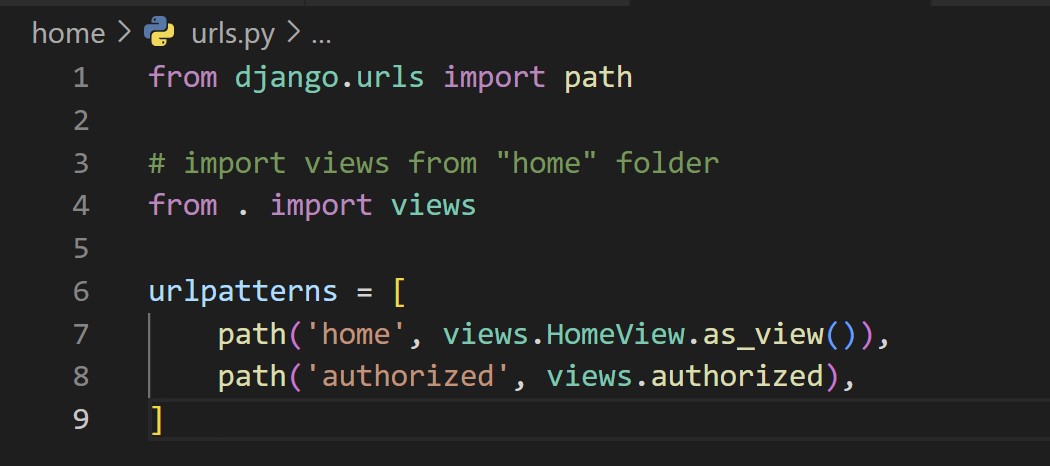
Let's go back to our code and change our views that are function-based to the ones that are class-based and see in detail how class-based views work. The first view we made was in the home app (views.py), so let's go back and change it. The only thing we need to do here is display a template. So let's in here from django.views.generic import TemplateView. Then, create a new class called HomeView that inherits from TemplateView, and the only thing we need to pass here is the ‘template\_name’.

We still need one more thing because our template requires some extra information, so we can add a variable called extra\_context and now pass this dictionary {'today': datetime.today()} in it.

So now we can delete ‘home’ function from here, and we have our first class-based views!



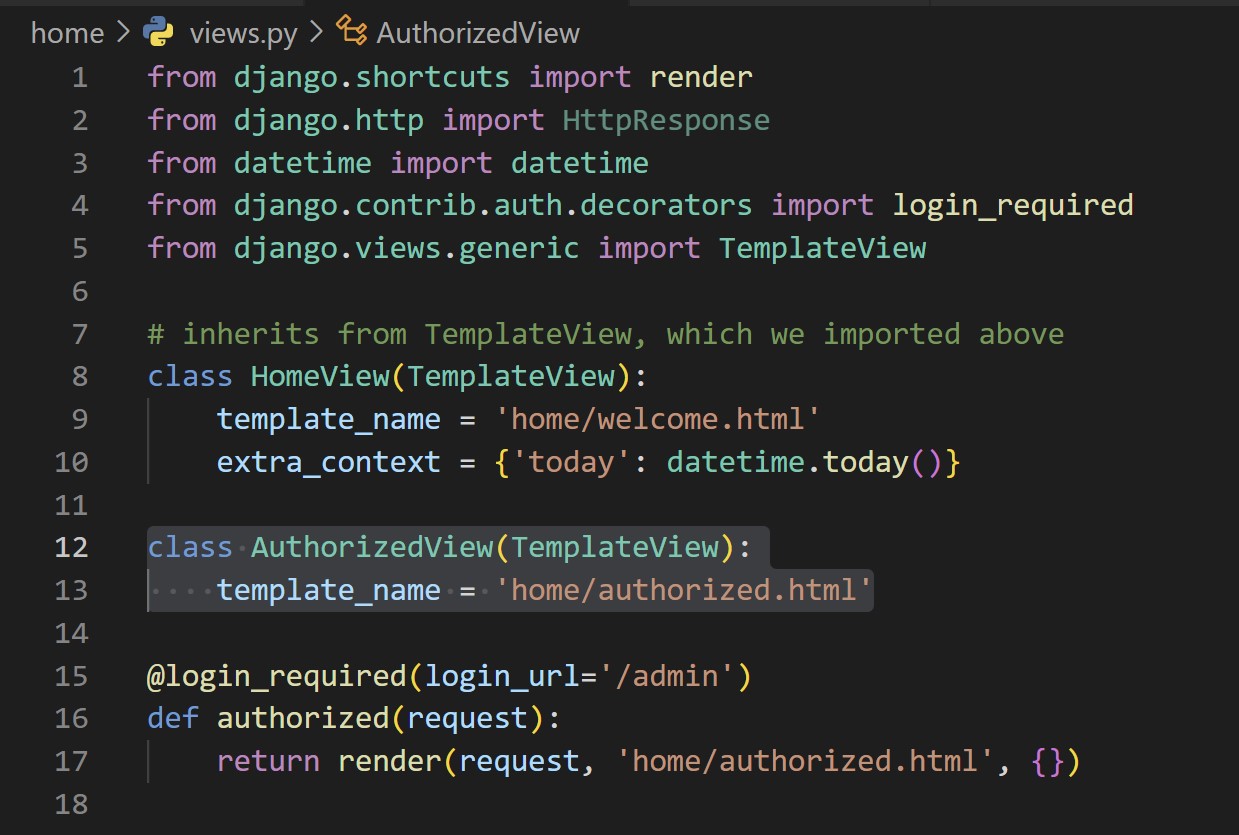
The last thing missing is that we need to change the way the URLs are defined, so let's go to the URLs, and instead of passing the home function, we're going to pass the HomeView class.



So we can do the same for the second function and create a class called AuthorizedView.

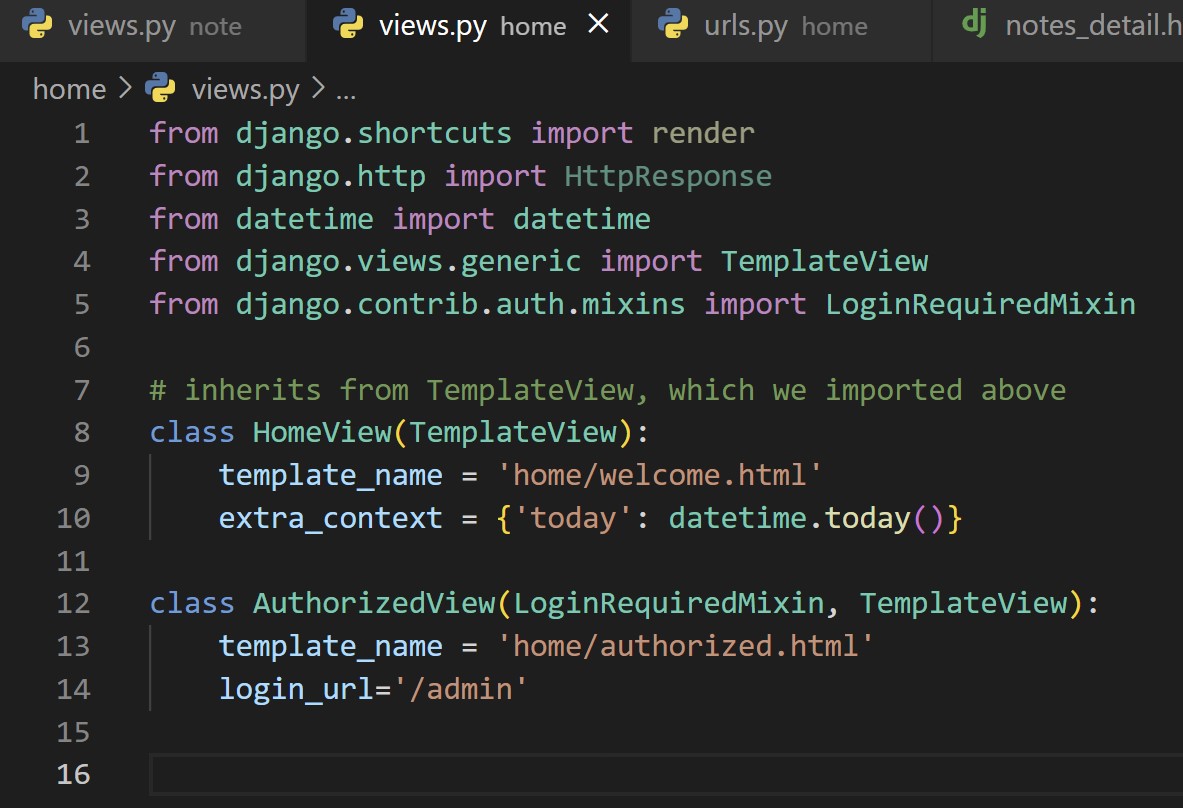
class AuthorizedView( TemplateView):

template\_name = 'home/authorized.html'



But we're still missing authentication. How do we handle authentication on class-based views? Well, to do that, we're going to need Mixins class. Mixins are helper classes that can be used along with other classes to provide additional features. For this case, we'll use the LoginRequiredMixin. The only thing we need to do here now is make sure that this class, which is Mixins, is added before the TemplateView in AuthorizedView parameters. The last thing is to add the login\_url.

So we can remove @login\_required now.



Now fix the URLs to be AuthorizedView.as\_view, and that's it.

Text

Description automatically generated

As you can see here, things are quite nice and well-organized, and you also don't have to remember the requests coming in and out of the function. Class-based views as the views increase in complexity, they become more and more amazing allies.