

# Passing Parameters to Templates.

---

Let's start with our extended template "projects.html" At this point, it should look like this:

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

<h1>Projects Template</h1>

<p>Eu Lorem nisi est eiusmod nulla proident fugiat officia sit. Sunt eiusmod
occaecat eiusmod nostrud Lorem fugiat aute qui adipisicing duis. Occaecat id
proident enim elit ea magna in pariatur non est Lorem culpa tempor
reprehenderit. Cupidatat dolore cupidatat mollit voluptate ipsum aliquip
deserunt anim aliquip cupidatat.</p>

<p>Et ipsum fugiat et quis nulla elit laborum culpa. Deserunt consectetur qui
aliqua occaecat sit ullamco cupidatat magna esse deserunt id ipsum dolore.
Aute adipisicing Lorem veniam nulla ullamco ipsum exercitation. Culpa quis
mollit anim irure pariatur do amet elit commodo eiusmod tempor laborum amet.
Est aliqua exercitation ex excepteur laborum aute quis sint nostrud sunt.
Irure et consectetur do consequat eiusmod et cupidatat ullamco ullamco sunt
laborum elit. Adipisicing consectetur nisi proident pariatur velit quis
mollit labore.</p>

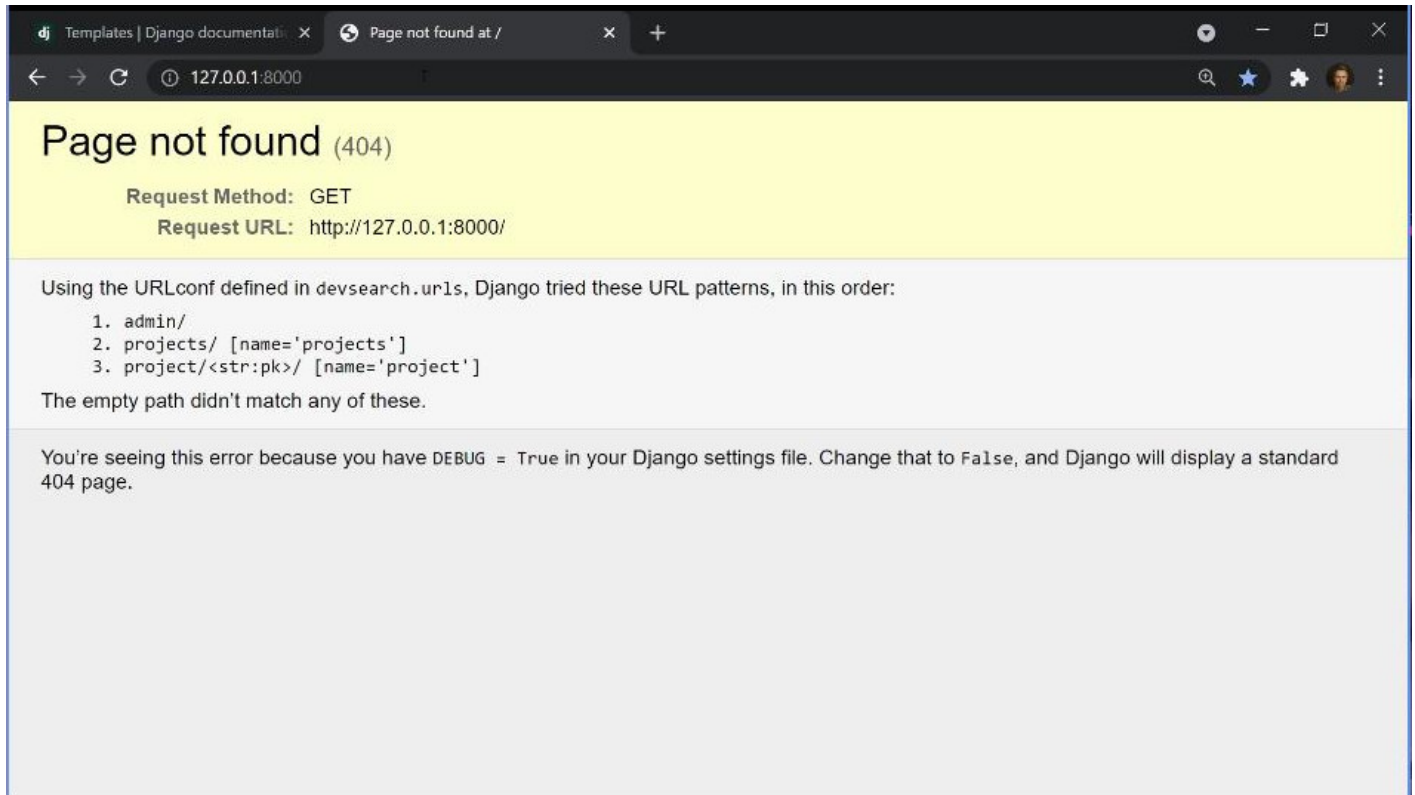
<p>Nostrud consectetur non in deserunt pariatur ipsum consectetur laboris
consequat cillum. Enim consequat laborum cillum pariatur aute non ea.
Excepteur cupidatat mollit aliqua commodo ad veniam velit in occaecat sunt
laborum cillum consequat.</p>

{% endblock content %}
```

For the time being, let's discuss about an error that you may be experiencing with your home page in this project. The error happens when you try to access your site's homepage when you type

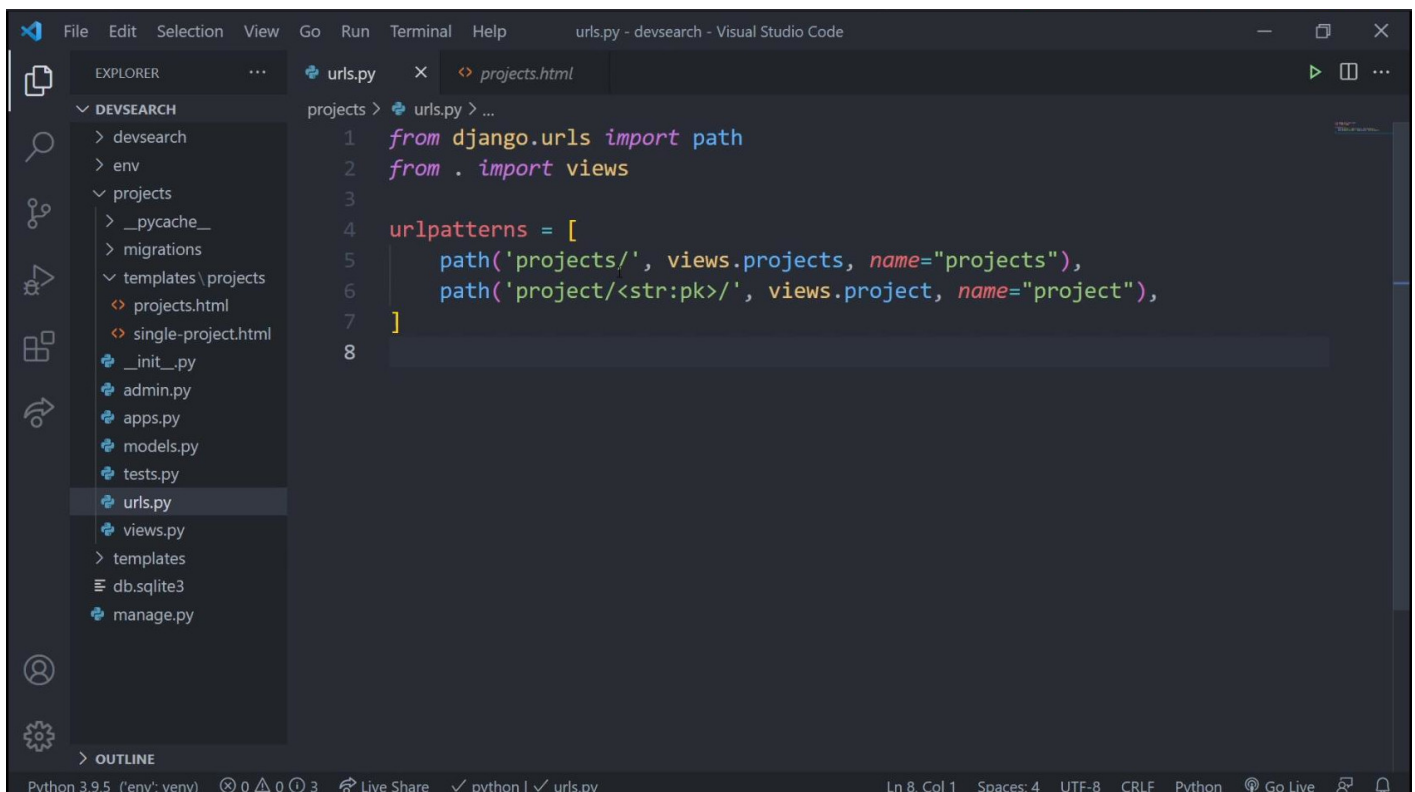
```
127.0.0.1:8000
```

in your browser

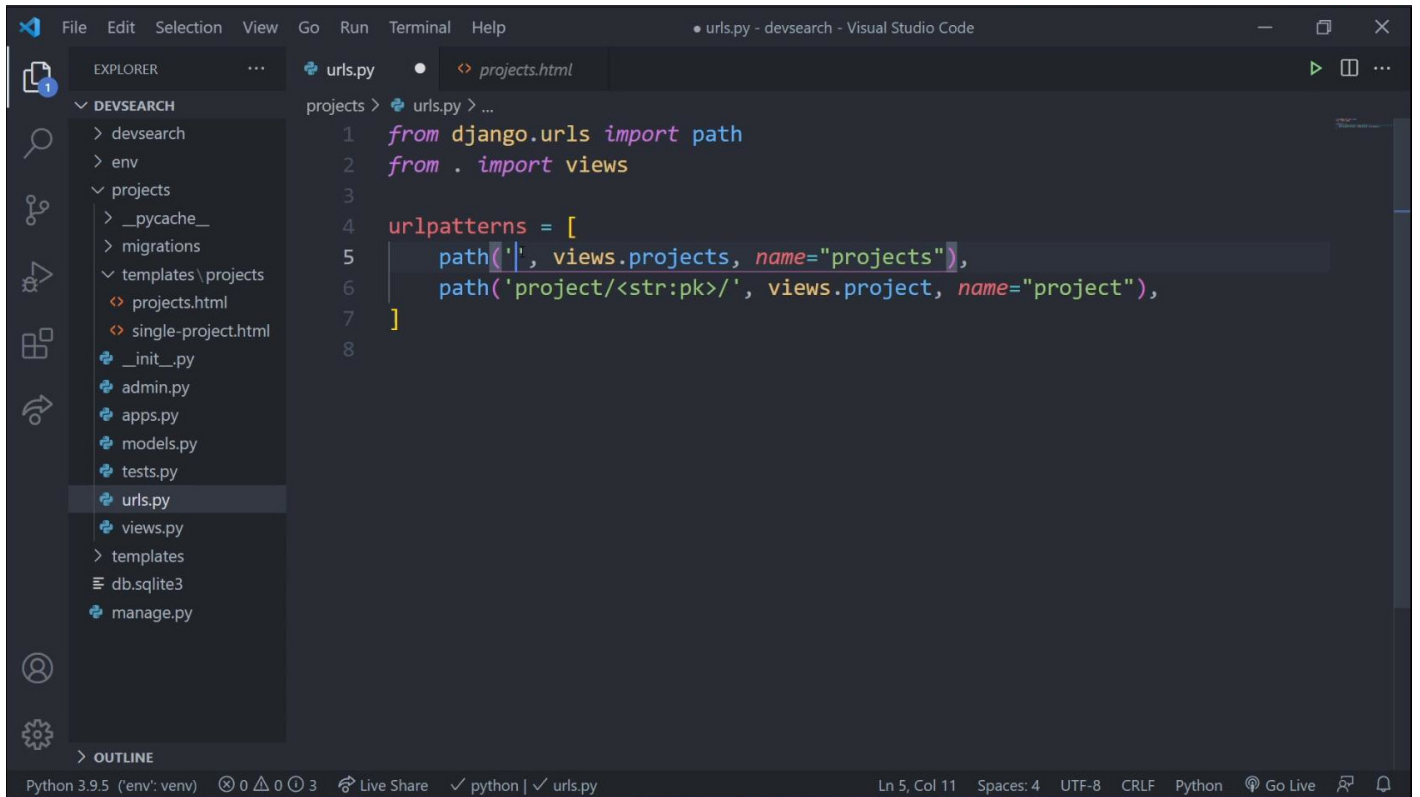


The reason for this error is the fact that we don't have a home page template yet. We have to patch this error by converting our projects.html into our home page for the moment.

In order to do this, let's access the projects/urls.py file:



and leave an empty string in place of 'projects/'



The screenshot shows the Visual Studio Code interface with a Django project named 'devsearch'. The Explorer sidebar on the left displays the project structure, including folders like 'devsearch', 'env', 'projects', and 'templates', and files like 'urls.py', 'views.py', and 'manage.py'. The 'urls.py' file is selected and open in the main editor. The code in 'urls.py' is as follows:

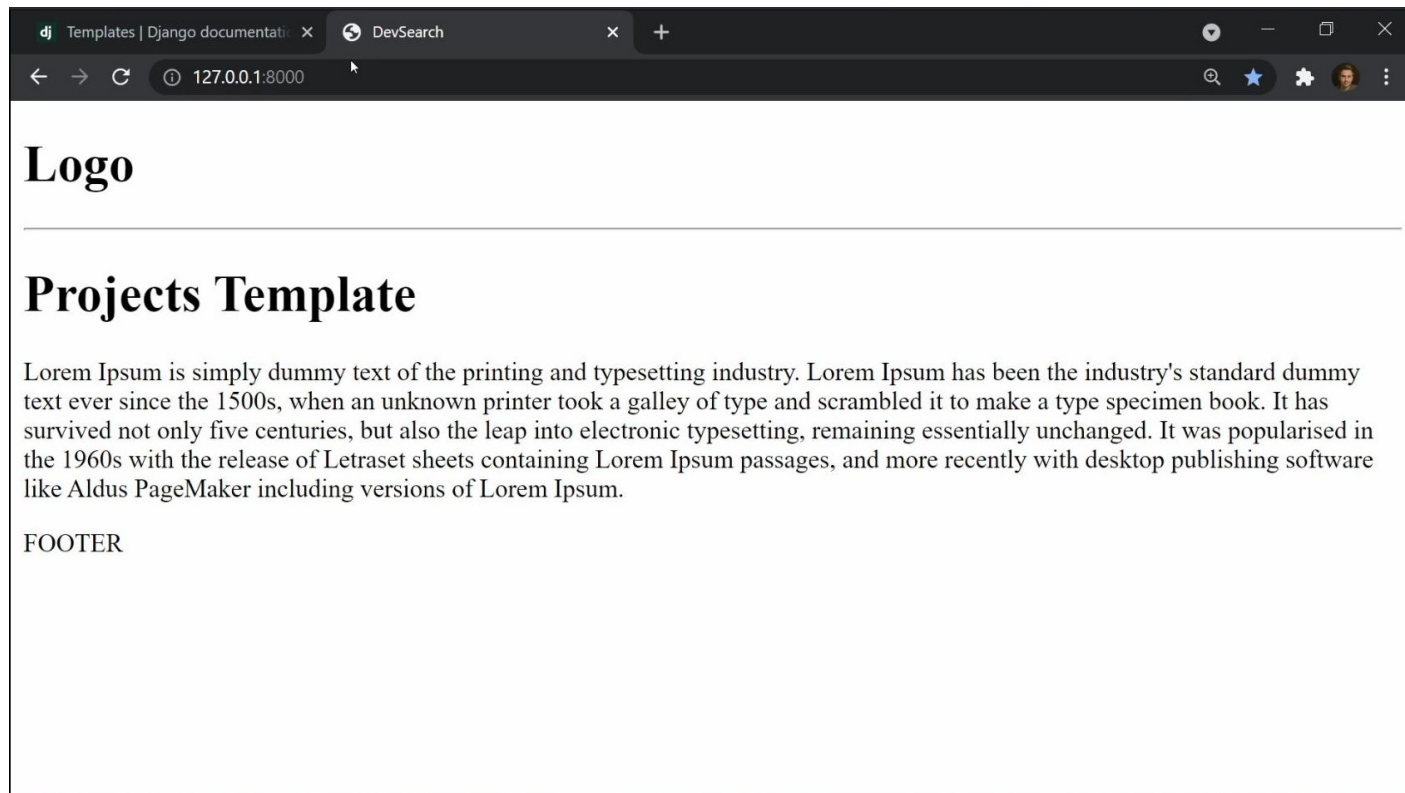
```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('', views.projects, name="projects"),
6     path('project/<str:pk>/', views.project, name="project"),
7 ]
8
```

The status bar at the bottom indicates the Python version is 3.9.5, the environment is 'env', and the file encoding is UTF-8.

Now if you open

127.0.0.1:8000

Our homepage should be our projects.html template



With that out of our way, let's go back to passing parameters to our templates.

Jinja, our template language, uses a "server" named template language server. This template "server" uses a different engine to compile the template and uses different place-holders to replace the text inside of them at compiling time.

Before explaining how to pass variables to our templates: `projects.html` and `project.html` we need to touch base on how Jinja template language handles various objects to interact with templates.

## Jinja Variables

Jinja uses `{{ named_variable }}` a double curly bracket to imply a variable "*named\_variable*" in this case, to signify the position of this variable inside our template.

For example, the statement

```
My first name is {{ first_name }} and my last name is {{ last_name }}
```

if we pass a "context" (a dictionary) with the variable place-holders such as:

```
{'first_name': 'James', 'last_name': 'Bond'}
```

The template will render:

```
My first name is James and my last name is Bond
```

Points to take home:

1. Variables are indicated by two nested pairs of curly braces `{{ variable_name }}`
2. You need a dictionary with `variable_name` as your key to access the variable inside the template and the value in the dictionary for the key `variable_name` will replace the contents inside the position occupied by the curly braces in the template.
3. Since you pass dictionaries, by implication, this means that you can also pass complete objects to the template:

```
Person('James', 'Bond')
```

then if the Person's object members have at least reading access, then you can replace the variables in the template as follows:

```
My name is {{ Person.last_name }}, {{ Person.first_name }} {{
Person.last_name}}.
```

and the template output will be:

```
My name is Bond, James Bond.
```

More details in [Django templates](#)

## Tags

Tags are the basic mechanism to insert python statements into the templates. Tags have kind of arbitrary logic to templates and their definition is quite vague. For example, tags can be used to output content, serve as a control structure in the **if** statement and **for**-loop control structure, grab content from a database or enable access to other template tags.

Tags are always surrounded by `{%, %}` pairs like this:

```
{% csrf_token %}
```

and they can accept arguments:

```
{% cycle 'odd' 'even' %}
```

Some tags require ending tags:

```
{% if user.authenticated %}  
    Hello {{ user.name }}!  
{% endif %}
```

or for filling lists or tables:

```
<ul>  
{% for n in user.names %}  
    <li> {{ n.last_name}}, {{ n.first_name }} </li>  
{% endfor %}  
</ul>
```

## Filters

Filters transform the values of variables and tag arguments.

They look like this:

```
{{ django|title }}
```

With a context of `{'django': 'the web framework for perfectionists with deadlines'}`, this template renders to:

The Web Framework For Perfectionists With Deadlines

Some filters take an argument:

```
{{ my_date|date:"Y-m-d" }}
```

## Comments

Comments look like this:

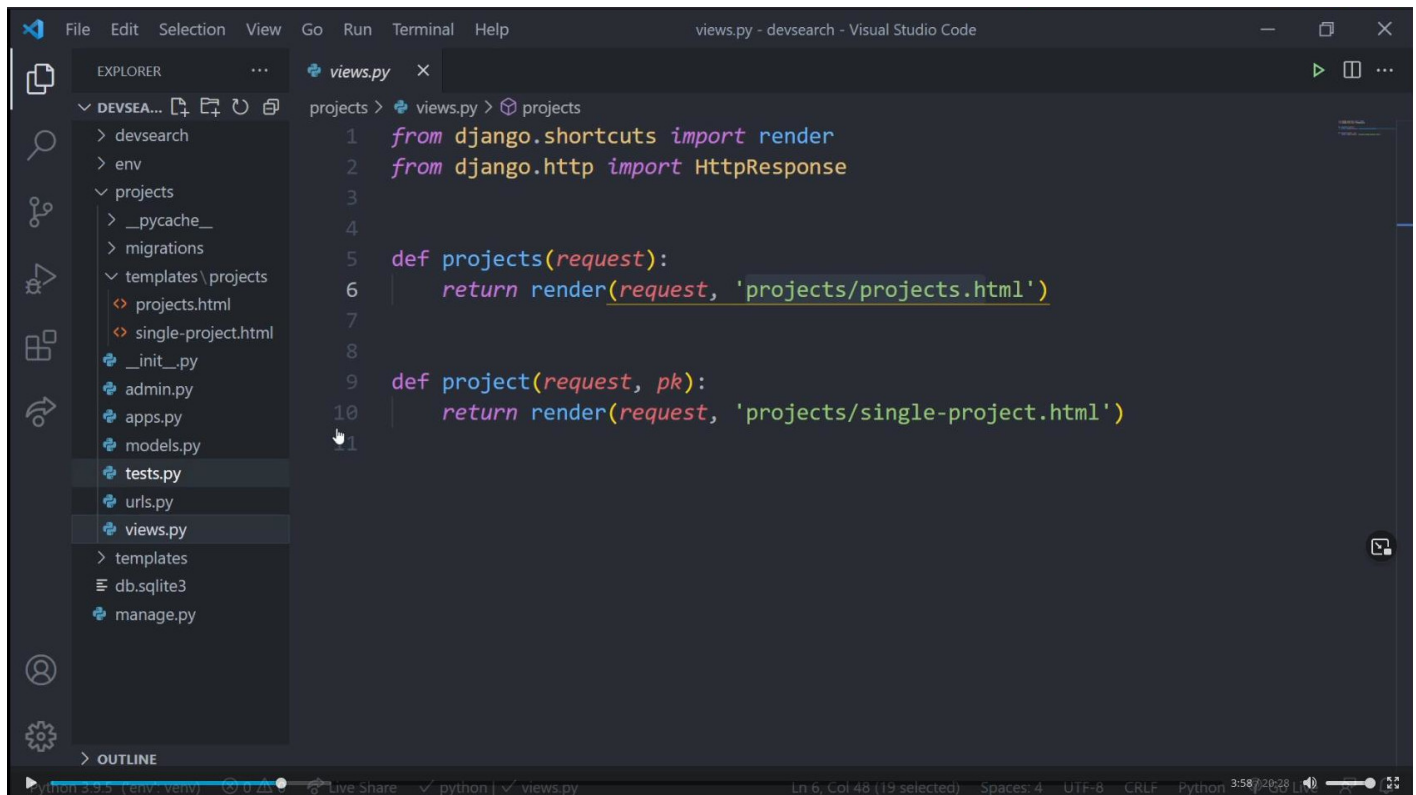
```
{# this won't be rendered #}
```

A {% comment %} tag provides multi-line comments.

## How to Pass Variables to Our Views

Once we have taken a look into how Jinja passes parameters and uses different control structures, lets modify our project views.

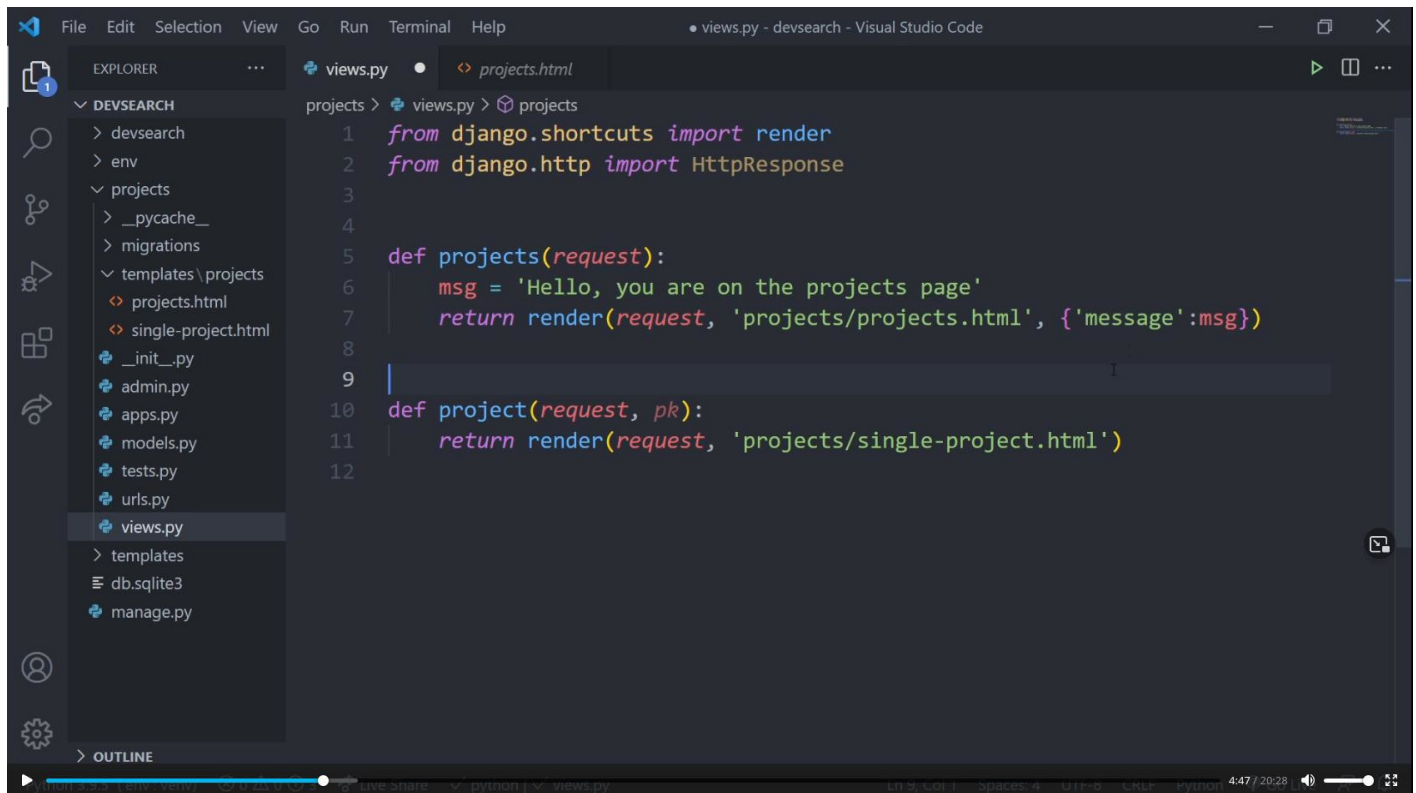
Open up the projects/views.py:



The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor in the center. The file explorer shows the project structure with the following files and folders: devsearch, env, projects, \_\_pycache\_\_, migrations, templates\projects, projects.html, single-project.html, \_\_init\_\_.py, admin.py, apps.py, models.py, tests.py, urls.py, views.py, templates, db.sqlite3, and manage.py. The code editor shows the contents of views.py, which includes imports for render and HttpResponseRedirect, and two view functions: projects and project.

```
1 from django.shortcuts import render
2 from django.http import HttpResponseRedirect
3
4
5 def projects(request):
6     return render(request, 'projects/projects.html')
7
8
9 def project(request, pk):
10    return render(request, 'projects/single-project.html')
```

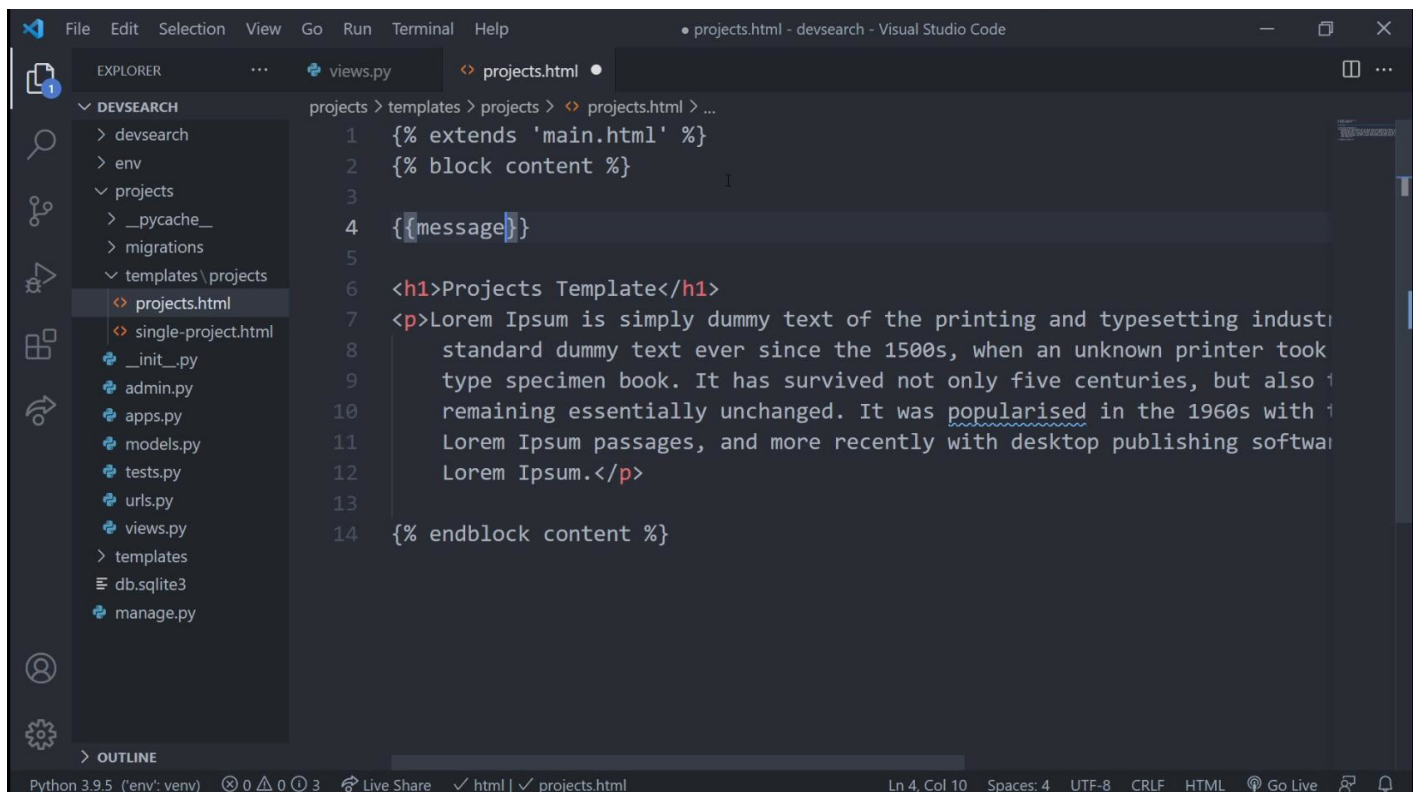
and change it to:



```
1 from django.shortcuts import render
2 from django.http import HttpResponseRedirect
3
4
5 def projects(request):
6     msg = 'Hello, you are on the projects page'
7     return render(request, 'projects/projects.html', {'message':msg})
8
9
10 def project(request, pk):
11     return render(request, 'projects/single-project.html')
```

Remember that the parameter is passed to the template via the dictionary 'key', so the variable name here is 'message'.

Next we need to modify our templates to accept the parameters we are passing to them from the views.py file



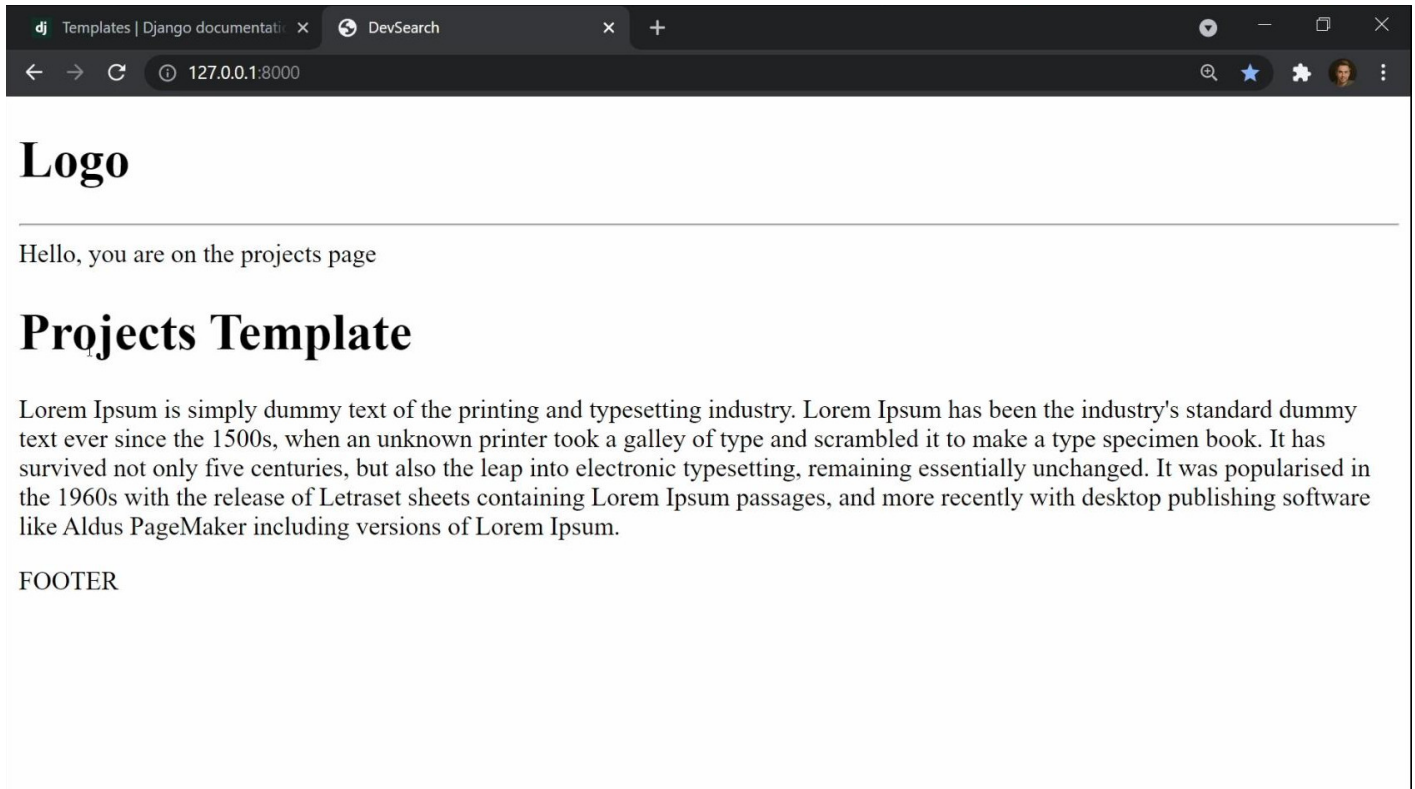
```
1 {% extends 'main.html' %}
2 {% block content %}
3
4     {{message}}
5
6     <h1>Projects Template</h1>
7     <p>Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker which generated Lorem Ipsum.</p>
8
9
10 {% endblock content %}
```

If we now run our server:

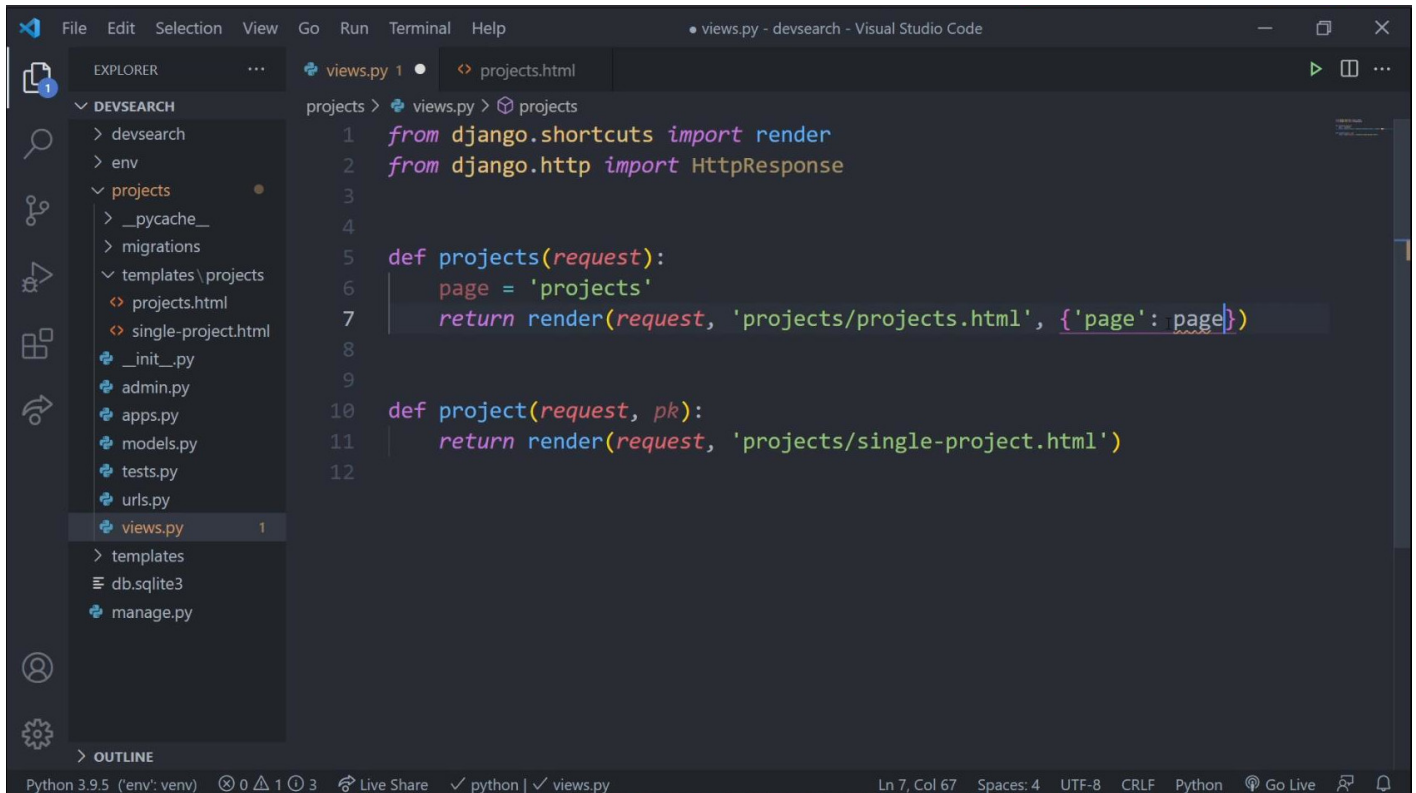


```
python manage.py runserver
```

and we access our home page at 127.0.0.1:8000, we'll see the following result:

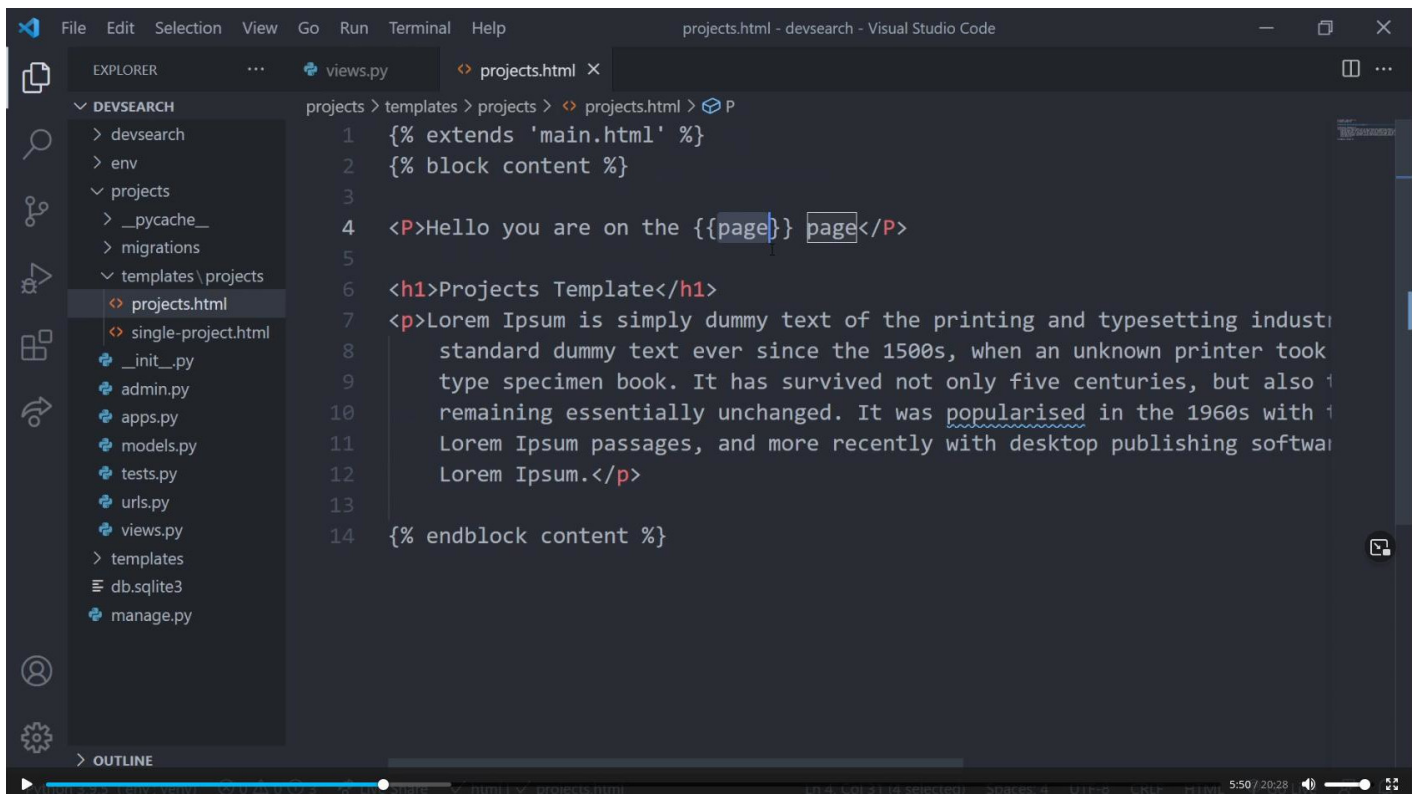


Now, let's modify the projects/views.py again to send another message:



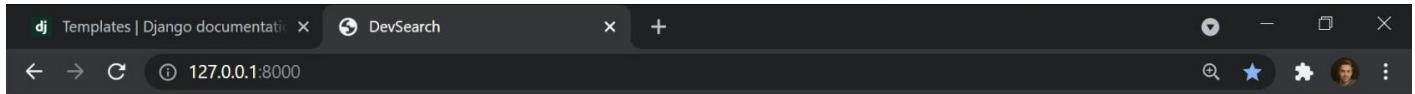
```
1 from django.shortcuts import render
2 from django.http import HttpResponseRedirect
3
4
5 def projects(request):
6     page = 'projects'
7     return render(request, 'projects/projects.html', {'page': page})
8
9
10 def project(request, pk):
11     return render(request, 'projects/single-project.html')
```

Here, the variable is now the key 'page'. And now, the template projects.html has also to be modified:



```
1 {% extends 'main.html' %}
2 {% block content %}
3
4 <P>Hello you are on the {{page}} page</P>
5
6 <h1>Projects Template</h1>
7 <p>Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.</p>
8
9
10
11
12
13
14 {% endblock content %}
```

should give you:



## Logo

Hello you are on the **projects** page

## Projects Template

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

FOOTER

### More Complex Parameters

Let's go back to our projects/views.py file. In there, let's insert a "database" with all the available projects which we would like to show in our view and pass to our template.

Let's add the following code after importing the libraries:

```
projectList = [
    {
        'id': '1',
        'title': 'E-commerce Website',
        'description': 'Fully functional e-commerce website'
    },
    {
        'id': '2',
        'title': 'Portfolio Website',
        'description': 'My fully functional portfolio website'
    },
    {
        'id': '3',
        'title': 'Chat and Secure Social Website',
        'description': 'Open source and secure communications client for social applications'
    },
    {
        'id': '4',
        'title': 'Project Management Website',
    },
]
```

```
        'description': 'Easy to use project management website for large
distributed teams'
    },
]
```

We also need to modify the views.py function projects to accept multiple parameters and we do that by changing the dictionary that provides the parameters to the template:

```
def projects(request):
    page = 'projects'
    context = {'page': page, 'projects': projectList}
    return render(request, 'projects/projects.html', context)
```

Please note that context is a dictionary that now contains all other parameters necessary for the projects.html template to render the page adequately.

The last step is to modify our projects/projects.html file to accept multiple parameters:

```
{% 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: {{ project.title }} --- {{ project.description }}
    </li>
    {% endfor %}
</ol>

{% endblock content %}
```

Run your server and check the page at your browser:

```
127.0.0.1:8000
```

This is because your home page is the projects page at this moment.

Now, lets implement the same changes for our project page. Change the definition for the views.py project function:

```
def project(request, pk):
    projectObj = None
    ids = [project.id for project in projectList]
    if pk not in ids:
        raise ValueError(f'Primary key: {pk} is not in the database')
    projectObj = projectList[ids.index(pk)]
    return render(request, 'projects/single-project.html', {'project':
projectObj})
```

and on the projects/single-project.html file:

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

{% block content %}

<h1> {{ project.title }} </h1>
<br>
<p> {{ project.description }} </p>

{% endblock content %}
```

test the project page using different slugs to access different contents:

```
127.0.0.1/project/3
127.0.0.1/project/1
127.0.0.1/project/4
127.0.0.1/project/2
```

and check the differences.

## Using References

We can also use self references to our own pages. In order to do this, we'll modify the projects/projects.html file to reference each project individually:

```
{% 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="/project/{{ project.id }}">{{
project.tile }}</a> --- {{ project.description }} </li>
    {% endfor %}
</ol>

{% endblock content %}

```

What we did was to add an anchor with a reference to the project/project.id page. This is how we can redirect to other pages inside the django project.

However, the code above is not easily maintainable. The reason being that any modification to the href="/project/{{project.id}}" would have to be propagate through the whole project, so this is not ideal for any possible changes to the slug.

A better way, 'the Django way' is to use tags as well, so the code should change to allow easy maintainence. This is the change:

```

{% 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 %}

```

Here

```
{% url '<template_name>' parameter %}
```

creates the required slug to access the specific project with the parameter id:

```
/project/{{parameter}}
```

This is called "url redirection."

In order for you to see the value of using the redirection tag {% url 'project' project.id %} lets do a simple experiment.

Go to the urls.py file in the projects folder and do the following change to the file:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.projects, name='projects'),
    path('project/<str:pk>', views.project, name="project")
]
```

change:

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

and save the change. And now run your server and take a look at the slug 127.0.0.1:8000 and click on the different links at the webpage and check if they are still working with the update.