Wexdi SoftSkills

Views and URL Routing

Writing function-based views

Wexdi SoftSkills

- In the world of Django, views play a crucial role in handling the flow of data between the application and the user.
- They define how a web page should respond to a user's request, and they are responsible for processing user input and returning a rendered template or an HTTP response.
- There are two primary ways to create views in Django: Function-Based Views (FBVs) and Class-Based Views (CBVs).
- Both approaches have their own advantages and limitations, and understanding these differences is essential for Django developers.
- In this post, we will discuss the key differences between FBVs and CBVs, provide examples of each, and help you determine which approach is the best fit for your project.

Writing function-based views

- Function-Based Views (FBVs)
  - Function-Based Views are the traditional and most straightforward way to create views in Django. As the name suggests, they are simply Python functions that take a request object as an argument and return an HTTP response. Let's take a look at a basic example of an FBV:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello, World!")
```

  - In this example, we define a view named hello, which takes a request object as its argument and returns an HTTP response with the text "Hello, World!". To wire this view to a URL, we would need to add it to our urls.py file like so:

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

urlpatterns = [
    path('hello/', views.hello, name='hello'),
]
```

Writing function-based views

- Function-Based Views (FBVs)
  - Advantages of FBVs:
    - Simplicity: FBVs are easy to understand, especially for beginners, as they are just Python functions.
    - Explicitness: The code in FBVs is more explicit, making it easier to trace the flow of data and understand what's happening at each step.
  - Limitations of FBVs:
    - Code repetition: As projects grow in complexity, you may find yourself repeating code across multiple views to handle similar functionality. This can make the codebase harder to maintain and update.
    - Less flexibility: FBVs can become unwieldy when dealing with complex views that require multiple methods or mixins to handle various aspects of the view's functionality.

Writing function-based views

- Class-Based Views (CBVs)
  - Class-Based Views, introduced in Django 1.3, provide an alternative approach to creating views. Instead of using functions, CBVs use Python classes and inherit from Django's built-in generic views to provide common functionalities. This approach can result in cleaner and more reusable code, especially for more complex applications. Let's take a look at a basic example of a CBV:

```
from django.http import HttpResponse
from django.views import View

class HelloView(View):
    def get(self, request):
        return HttpResponse("Hello, World!")
```

  - In this example, we define a view named HelloView that inherits from Django's base View class. We then define a get method to handle the GET request and return an HTTP response with the text "Hello, World!". To wire this view to a URL, we would need to add it to our urls.py file like so:

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

urlpatterns = [
    path('hello/', views.HelloView.as_view(), name='hello'),
]
```

Writing function-based views

- Class-Based Views (CBVs)
  - Advantages of CBVs:
    - Reusability: CBVs promote code reusability through inheritance, allowing you to create a common base class for views with shared functionalities.
    - Modularity: CBVs make it easier to organize and structure complex views by breaking them into smaller, more manageable pieces (methods and mixins).
    - Built-in generic views: Django comes with a set of generic views that provide commonly used functionality, such as creating, updating, and deleting objects, simplifying the development process.
  - Limitations of CBVs:
    - Learning curve: CBVs can be more challenging for beginners to grasp initially due to their reliance on inheritance and mixins.
    - Less explicit: The flow of data in CBVs can be less clear than in FBVs, as it may involve multiple layers of inheritance, making it harder to understand what's happening at each step.

Dynamic URL patterns.

- This module needs to be included in the root URLConf module as follows:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('books.urls')),
]
```

This tells Django to search for URL patterns in the file books/urls.py.

URL patterns

Here's a sample code for books/urls.py:

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

urlpatterns = [
    path('books/<int:pk>/', views.book_detail),
    path('books/<str:genre>/', views.books_by_genre),
    path('books/', views.book_index),
]
```

Django's HttpResponse and render methods.

Wexdi SoftSkills

- **Render**
  - This is arguments for render. It takes the template(template_name) and combines with a given context dictionary and returns an HttpResponse object with that rendered text.

    Syntax:

    render(request, template_name, context=None, content_type=None, status=None, using=None)

    #With render

```
def view_page(request):
    # View code here...
    return render(request, 'app/index.html', {
        'value': 'data',
    }, content_type='application/xhtml+xml')
```

- **HttpRespone**
  - Even render return HttpResponse but it can render the template with the context(If a value in the dictionary is callable, the view will call it just before rendering the template.)

```
#with HttpResponse
def view_page(request):
    # View code here...
    t = loader.get_template('app/index.html')
    c = {'value': 'data'}
    return HttpResponse(t.render(c, request), content_type='application/xhtml+xml')
```