Learn Python with Django

Generic list and detail

Overview

The process is similar to the creating the index page, which we showed in the previous tutorial. We'll still need to create URL maps, views and templates. The main difference is that for the detail pages we'll have the additional challenge of extracting information from patterns in the URL and passing it to the view. For these pages we're going to demonstrate a completely different type of view: generic class-based list and detail views. These can significantly reduce the amount of view code needed, making them easier to write and maintain.

The final part of the tutorial will demonstrate how to paginate your data when using generic class-based list views.

Book list page

The book list page will display a list of all the available book records in the page, accessed using the url: `catalog/books/`. The page will display a title and author for each record, with the title being a hyperlink to the associated book detail page. The page will have the same structure and navigation as all other pages in the site, and we can therefore extend the base template (**base_generic.html**) we created in the previous tutorial.

URL mapping

Open **/catalog/urls.py** and copy in the line shown in bold below. As for the index page, this `path()` function defines a pattern to match against the URL (**'books/'**), a view function that will be called if the URL matches (`views.BookListView.as_view()`) , and a name for this particular mapping.

```
from django.conf.urls import url

from . import views
from django.urls import path

urlpatterns = [
    path('', views.index, name='index'),
    path('books/', views.BookListView.as_view(), name='books'),
    path('book/<int:pk>', views.BookDetailView.as_view(), name='book-detail'),
]
```

As discussed in the previous tutorial the URL must already have matched `/catalog`, so the view will actually be called for the URL: `/catalog/books/`.

The view function has a different format than before — that's because this view will actually be implemented as a class. We will be inheriting from an existing generic view function that already does most of what we want this view function to do, rather than writing our own from scratch.

For Django class-based views we access an appropriate view function by calling the class method `as_view()`. This does all the work of creating an instance of the class, and making sure that the right handler methods are called for incoming HTTP requests.

View (class-based)

We could quite easily write the book list view as a regular function (just like our previous index view), which would query the database for all books, and then call `render()` to pass the list to a specified template. Instead however, we're going to use a class-based generic list view (`ListView`) — a class that inherits from an existing view. Because the generic view already implements most of the functionality we need, and follows Django best-practice, we will be able to create a more robust list view with less code, less repetition, and ultimately less maintenance.

Open **catalog/views.py**, and copy the following code into the bottom of the file:

```
from django.views import generic

class BookListView(generic.ListView):
    model = Book
```

That's it! The generic view will query the database to get all records for the specified model (`Book`) then render a template located at **/locallibrary/catalog/templates/catalog/book_list.html** (which we will create below). Within the template you can access the list of books with the template variable named `object_list` OR `book_list` (i.e. generically "*the_model_name*`_list`").

You can add attributes to change the default behaviour above. For example, you can specify another template file if you need to have multiple views that use this same model, or you might want to use a different template variable name if `book_list` is not intuitive for your particular template use-case. Possibly the most useful variation is to change/filter the subset of results that are returned — so instead of listing all books you might list top 5 books that were read by other users.

```
class BookListView(generic.ListView):
    model = Book
    context_object_name = 'my_book_list'   # your own name for the list as a template
variable
    queryset = Book.objects.filter(title__icontains='war')[:5] # Get 5 books containing
the title war
    template_name = 'books/my_arbitrary_template_name_list.html'  # Specify your own
template name/location
```

Overriding methods in class-based views

While we don't need to do so here, you can also override some of the class methods.

For example, we can override the `get_queryset()` method to change the list of records returned. This is more flexible than just setting the `queryset` attribute as we did in the preceding code fragment (though there is no real benefit in this case):

```
class BookListView(generic.ListView):
    model = Book

    def get_queryset(self):
        return Book.objects.filter(title__icontains='war')[:5] # Get 5 books containing
the title war
```

We might also override `get_context_data()` in order to pass additional context variables to the template (e.g. the list of books is passed by default). The fragment below shows how to add a variable named "`some_data`" to the context (it would then be available as a template variable).

```
class BookListView(generic.ListView):
    model = Book

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(BookListView, self).get_context_data(**kwargs)
        # Get the blog from id and add it to the context
        context['some_data'] = 'This is just some data'
        return context
```

When doing this it is important to follow the pattern used above:

- First get the existing context from our superclass.
- Then add your new context information.
- Then return the new (updated) context.

Creating the List View template

Create the HTML file **/locallibrary/catalog/templates/catalog/book_list.html** and copy in the text below. As discussed above, this is the default template file expected by the generic class-based list view (for a model named `Book` in an application named `catalog`).

Templates for generic views are just like any other templates (although of course the context/information passed to the template may differ). As with our *index* template, we extend our base template in the first line, and then replace the block named `content`.

```
{% extends "base_generic.html" %}

{% block content %}
    <h1>Book List</h1>

    {% if book_list %}
    <ul>

      {% for book in book_list %}
      <li>
        <a href="{{ book.get_absolute_url }}">{{ book.title }}</a> ({{book.author}})
      </li>
      {% endfor %}

    </ul>
    {% else %}
      <p>There are no books in the library.</p>
    {% endif %}
{% endblock %}
```

The view passes the context (list of books) by default as `object_list` and `book_list` aliases; either will work.

Conditional execution

We use the `if`, `else` and `endif` template tags to check whether the `book_list` has been defined and is not empty. If `book_list` is empty, then the `else` clause displays text explaining that there are no books to list. If `book_list` is not empty, then we iterate through the list of books.

```
{% if book_list %}
  <!-- code here to list the books -->
{% else %}
  <p>There are no books in the library.</p>
```

```
{% endif %}
```

The condition above only checks for one case, but you can test on additional conditions using the `elif` template tag (e.g. `{% elif var2 %}`).

For loops

The template uses the for and `endfor` template tags to loop through the book list, as shown below. Each iteration populates the `book` template variable with information for the current list item.

```
{% for book in book_list %}
  <li> <!-- code here get information from each book item --> </li>
{% endfor %}
```

While not used here, within the loop Django will also create other variables that you can use to track the iteration. For example, you can test the `forloop.last` variable to perform conditional processing the last time that the loop is run.

Accessing variables

The code inside the loop creates a list item for each book that shows both the title (as a link to the yet-to-be-created detail view) and the author.

```
<a href="{{ book.get_absolute_url }}">{{ book.title }}</a> ({{book.author}})
```

We access the *fields* of the associated book record using the "dot notation" (e.g. `book.title` and `book.author`), where the text following the `book` item is the field name (as defined in the model).

We can also call *functions* in the model from within our template — in this case we call `Book.get_absolute_url()` to get an URL you could use to display the associated detail record. This works provided the function does not have any arguments (there is no way to pass arguments!)

Update the base template

Open the base template (**/locallibrary/catalog/templates/*base_generic.html***) and insert **{% url 'books' %}** into the URL link for **All books**, as shown below. This will enable the link in all pages (we can successfully put this in place now that we've created the "books" url mapper).

```
<li><a href="{% url 'index' %}">Home</a></li>
<li><a href="{% url 'books' %}">All books</a></li>
<li><a href="">All authors</a></li>
```

What does it look like?

You won't be able to build book list yet, because we're still missing a dependency — the URL map for the book detail pages, which is needed to create hyperlinks to individual books. We'll show both list and detail views after the next section.

Book detail page

The book detail page will display information about a specific book, accessed using the URL `catalog/book/<id>` (where *<id>* is the primary key for the book). In addition to fields in the `Book` model (author, summary, ISBN, language, and genre), we'll also list the details of the available copies (`BookInstances`) including the status, expected return date, imprint, and id. This will allow our readers not just to learn about the book, but also to confirm whether/when it is available.

URL mapping

Open **/catalog/urls.py** and add the '**book-detail**' URL mapper shown in bold below. This `path()` function defines a pattern, associated generic class-based detail view, and a name.

```
urlpatterns = [
    path('', views.index, name='index'),
    path('books/', views.BookListView.as_view(), name='books'),
    path('book/<int:pk>', views.BookDetailView.as_view(), name='book-detail'),
]
```

For the *book-detail* path the URL pattern uses a special syntax to capture the specific id of the book that we want to see. The syntax is very simple: angle brackets define the part of the URL to be captured, enclosing the name of the variable that the view can use to access the captured data. For example, **<something>** , will capture the marked pattern and pass the value to the view as a variable "something". You can optionally precede the variable name with a converter specification that defines the type of data (int, str, slug, uuid, path).

In this case we use `'<int:pk>'` to capture the book id, which must be an integer, and pass it to the view as a parameter named `pk` (short for primary key).

Passing additional options in your URL maps

One feature that we haven't used here, but which you may find valuable, is that you can declare and pass additional options to the view. The options are declared as a dictionary that you pass as the third un-named argument to the `path()` function. This approach can be useful if you want to use the same view for multiple resources, and pass data to configure its behaviour in each case (below we supply a different template in each case).

```
path('url/', views.my_reused_view, {'my_template_name': 'some_path'}, name='aurl'),
path('anotherurl/', views.my_reused_view, {'my_template_name': 'another_path'},
name='anotherurl'),
```

View (class-based)

Open **catalog/views.py**, and copy the following code into the bottom of the file:

```
class BookDetailView(generic.DetailView):
    model = Book
```

That's it! All you need to do now is create a template called **/locallibrary/catalog/templates/catalog/book_detail.html**, and the view will pass it the database information for the specific `Book` record extracted by the URL mapper. Within the template you can access the list of books with the template variable named `object` OR `book` (i.e. generically "*the_model_name*").

If you need to, you can change the template used and the name of the context object used to reference the book in the template. You can also override methods to, for example, add additional information to the context.

What happens if the record doesn't exist?

If a requested record does not exist then the generic class-based detail view will raise an `Http404` exception for you automatically — in production this will automatically display an appropriate "resource not found" page, which you can customize if desired.

Just to give you some idea of how this works, the code fragment below demonstrates how you would implement the class-based view as a function, if you were **not** using the generic class-based detail view.

```
def book_detail_view(request,pk):
    try:
        book_id=Book.objects.get(pk=pk)
    except Book.DoesNotExist:
        raise Http404("Book does not exist")

    #book_id=get_object_or_404(Book, pk=pk)

    return render(
        request,
        'catalog/book_detail.html',
        context={'book':book_id,}
    )
```

The view first tries to get the specific book record from the model. If this fails the view should raise an `Http404` exception to indicate that the book is "not found". The final step is then, as usual, to call `render()` with the template name and the book data in the `context` parameter (as a dictionary).

Creating the Detail View template

Create the HTML file **/locallibrary/catalog/templates/catalog/book_detail.html** and give it the below content. As discussed above, this is the default template file name expected by the generic class-based *detail* view (for a model named `Book` in an application named `catalog`).

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Title: {{ book.title }}</h1>

  <p><strong>Author:</strong> <a href="">{{ book.author }}</a></p> <!-- author detail
link not yet defined -->
  <p><strong>Summary:</strong> {{ book.summary }}</p>
  <p><strong>ISBN:</strong> {{ book.isbn }}</p>
  <p><strong>Language:</strong> {{ book.language }}</p>
  <p><strong>Genre:</strong> {% for genre in book.genre.all %} {{ genre }}{% if not
forloop.last %}, {% endif %}{% endfor %}</p>

  <div style="margin-left:20px;margin-top:20px">
    <h4>Copies</h4>

    {% for copy in book.bookinstance_set.all %}
    <hr>
    <p class="{% if copy.status == 'a' %}text-success{% elif copy.status == 'm' %}text-
danger{% else %}text-warning{% endif %}">{{ copy.get_status_display }}</p>
    {% if copy.status != 'a' %}<p><strong>Due to be returned:</strong>
{{copy.due_back}}</p>{% endif %}
    <p><strong>Imprint:</strong> {{copy.imprint}}</p>
    <p class="text-muted"><strong>Id:</strong> {{copy.id}}</p>
    {% endfor %}
  </div>
{% endblock %}
```

The author link in the template above has an empty URL because we've not yet created an author detail page. Once that exists, you should update the URL like this:

```
<a href="{% url 'author-detail' book.author.pk %}">{{ book.author }}</a>
```

Though a little larger, almost everything in this template has been described previously:

- We extend our base template and override the "content" block.
- We use conditional processing to determine whether or not to display specific content.
- We use `for` loops to loop through lists of objects.
- We access the context fields using the dot notation (because we've used the detail generic view, the context is named `book`; we could also use "`object`")

The one interesting thing we haven't seen before is the function `book.bookinstance_set.all()`. This method is "automagically" constructed by Django in order to return the set of `BookInstance` records associated with a particular `Book`.

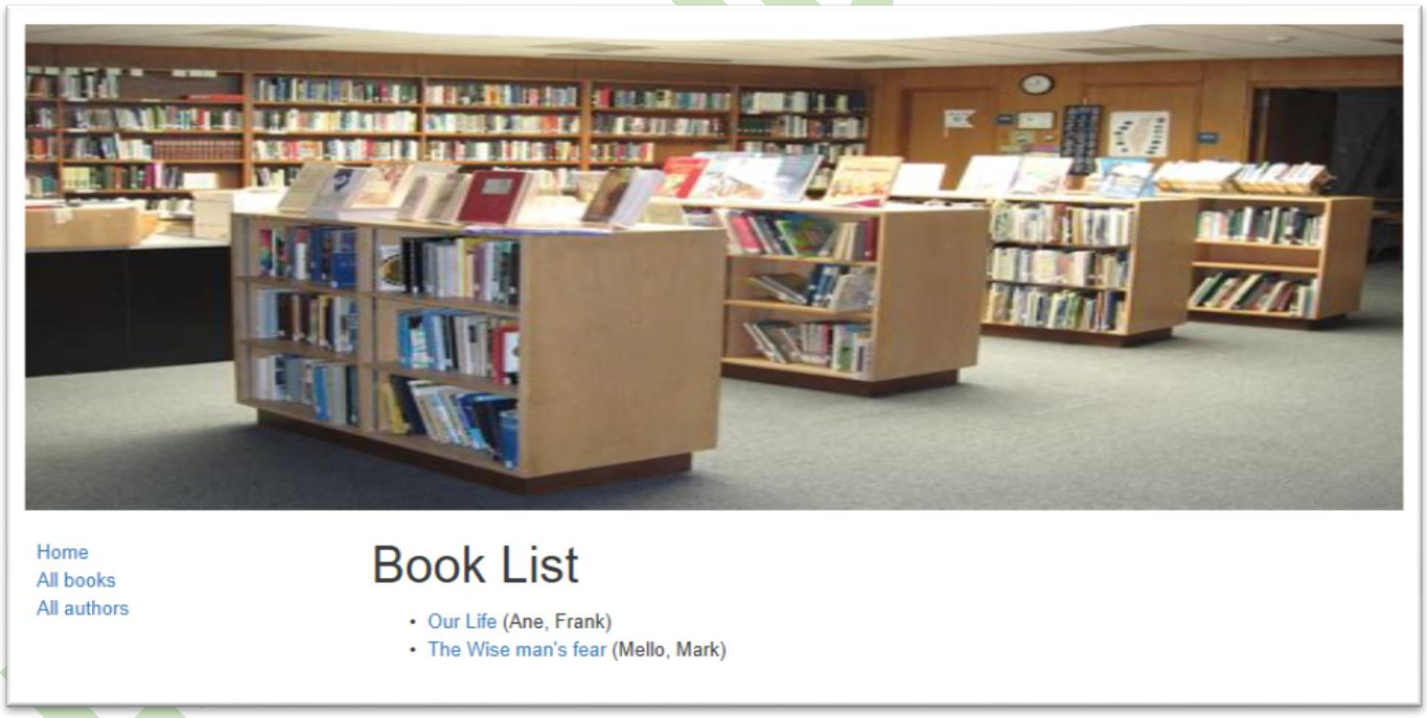
```
{% for copy in book.bookinstance_set.all %}
<!-- code to iterate across each copy/instance of a book -->
{% endfor %}
```

This method is needed because you declare a `ForeignKey` (one-to many) field in only the "one" side of the relationship. Since you don't do anything to declare the relationship in the other ("many") model, it doesn't have any field to get the set of associated records. To overcome this problem, Django constructs an appropriately named "reverse lookup" function that you can use. The name of the function is constructed by lower-casing the model name where the `ForeignKey` was declared, followed by `_set` (i.e. so the function created in `Book` is `bookinstance_set()`).

What does it look like?

At this point we should have created everything needed to display both the book list and book detail pages. Run the server (`python3 manage.py runserver`) and open your browser to http://127.0.0.1:8000/.

Click the **All books** link to display the list of books.



Then click a link to one of your books. If everything is set up correctly, you should see something like the following screenshot.

Pagination

If you've just got a few records, our book list page will look fine. However, as you get into the tens or hundreds of records the page will take progressively longer to load (and have far too much content to browse sensibly). The solution to this problem is to add pagination to your list views, reducing the number of items displayed on each page.

Django has excellent in-built support for pagination. Even better, this is built into the generic class-based list views so you don't have to do very much to enable it!

Views

Open **catalog/views.py**, and add the `paginate_by` line shown in bold below.

```
class BookListView(generic.ListView):
    model = Book
    paginate_by = 10
```

With this addition, as soon as you have more than 10 records the view will start paginating the data it sends to the template. The different pages are accessed using GET parameters — to access page 2 you would use the URL: `/catalog/books/?page=2`.

Templates

Now that the data is paginated, we need to add support to the template to scroll through the results set. Because we might want to do this in all list views, we'll do this in a way that can be added to the base template.

Open ***/locallibrary/catalog/templates/base_generic.html*** and copy in the following pagination block below our content block (highlighted below in bold). The code first checks if pagination is enabled on the current page. If so then it adds next and previous links as appropriate (and the current page number).

```
{% block content %}{% endblock %}

{% block pagination %}
  {% if is_paginated %}
      <div class="pagination">
          <span class="page-links">
              {% if page_obj.has_previous %}
                  <a href="{{ request.path }}?page={{ page_obj.previous_page_number
}}">previous</a>
              {% endif %}
              <span class="page-current">
                  Page {{ page_obj.number }} of {{ page_obj.paginator.num_pages }}.
              </span>
              {% if page_obj.has_next %}
```

```
                    <a href="{{ request.path }}?page={{ page_obj.next_page_number
}}">next</a>
                {% endif %}
            </span>
        </div>
    {% endif %}
{% endblock %}
```

The `page_obj` is a Paginator object that will exist if pagination is being used on the current page. It allows you to get all the information about the current page, previous pages, how many pages there are, etc.

We use `{{ request.path }}` to get the current page URL for creating the pagination links. This is useful, because it is independent of the object that we're paginating.

Thats it!

What does it look like?

The screenshot below shows what the pagination looks like — if you haven't entered more than 10 titles into your database, then you can test it more easily by lowering the number specified in the `paginate_by` line in your **catalog/views.py** file. To get the below result we changed it to `paginate_by = 2`.

The pagination links are displayed on the bottom, with next/previous links being displayed depending on which page you're on.