Django Tutorial Part 6: Generic list and detail views

Created by 이 윤 준 (yoonjoon.lee@gmail.com)

May, 2019

This extends our LocalLibrary website, adding list and detail pages for books and authors. Here we'll

- learn about generic class-based views, and
- show how they can reduce the amount of code we have to write for common use cases.
- go into URL handling in greater detail, showing how to perform basic pattern matching.

Overview

We're going to complete the first version of the LocalLibrary website by adding list and detail pages for books and authors.

To be more precise, we'll show you how to implement the book pages, and get you to create the author pages yourself!

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.

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**).

URL mapping

Open **/catalog/urls.py** and copy in the line. 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.

```
urlpatterns = [
    path('', views.index, name='index'),
    path('books/', views.BookListView.as_view(), name='books'),
]
```

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.

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 code into the bottom of the file:

```
from django.views import generic

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

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**. Within the template we can access the list of books with the template variable named `object_list` OR `book_list` (i.e. generically "`the_model_name_list`").

We can add attributes to change the default behaviour above. For example, we can specify another template file if we 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 our 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.

Overriding methods in class-based views

We 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:

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

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

We might also override `get_context_data()` in order to pass additional context variables to the template. The fragment shows how to add a variable named "some_data" to the context.

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

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get the co
        context = super(BookListView, self).
            get_context_data(**kwargs)
        # Create any data 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. As discussed above, this is the default template file expected by the generic class-based list view.

Templates for generic views are just like any other templates. 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 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. 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 %}
```

Within the loop Django will also create other variables that we can use to track the iteration. For example, we 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 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", where the text following the book item is the field name.

We can also call functions in the model from within our template — we call `Book.get_absolute_url()` to get an URL we could use to display the associated detail record. This works does not have any 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**. 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?

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.

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 us 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. 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='boo
    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. We 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 a specially formatted string and pass it to the view as a parameter named `pk`. This is the id that is being used to store the book uniquely in the database, as defined in the Book Model.

Advanced path matching/regular expression primer

The pattern matching provided by `path()` is simple and useful for the cases where we just want to capture *any* string or integer. If we need more refined filtering (for example, to filter only strings that have a certain number of characters) then we can use the `re_path()` method.

This method is used just like `path()` except that it allows you to specify a pattern using a Regular expression.

```
re_path(
    r'^book/(?P<pk>\d+)$',
    views.BookDetailView.as_view(),
    name='book-detail')
```

Regular expressions are an incredibly powerful pattern mapping tool. They are, frankly, quite unintuitive and scary for beginners.

The first thing to know is that regular expressions should usually be declared using the raw string literal syntax (i.e. they are enclosed as shown: **r'<your regular expression text goes here>'**).

Symbol	Meaning
^	Match the beginning of the text
$	Match the end of the text
\d	Match a digit (0, 1, 2, ... 9)
\w	Match a word character, e.g. any upper- or lower-case character in the alphabet, digit or the underscore character (_)
+	Match one or more of the preceding character. For example, to match one or more digits you would use `\d+`. To match one or more "a" characters, you could use `a+`
*	Match zero or more of the preceding character. For example, to match nothing or a word you could use `\w*`
( )	Capture the part of the pattern inside the brackets. Any captured values will be passed to the view as unnamed parameters (if multiple patterns are captured, the associated parameters will be supplied in the order that the captures were declared).
(?P<*name*>...)	Capture the pattern (indicated by ...) as a named variable (in this case "name"). The captured values are passed to the view with the name specified. Your view must therefore declare an argument with the same name!
[ ]	Match against one character in the set. For example, [abc] will match on 'a' or 'b' or 'c'. [-\w] will match on the '-' character or any word character.

A few real examples of patterns:

Pattern	Description
r'^book/(?P<pk>\d+)$'	This is the RE used in our URL mapper. It matches a string that has `book/` at the start of the line (**^book/**), then has one or more digits (`\d+`), and then ends (with no non-digit characters before the end of line marker). It also captures all the digits **(?P<pk>\d+)** and passes them to the view in a parameter named 'pk'. **The captured values are always passed as a string!** For example, this would match `book/1234` , and send a variable `pk='1234'` to the view.
r'^book/(\d+)$'	This matches the same URLs as the preceding case. The captured information would be sent as an unnamed argument to the view.
r'^book/(?P<stub>[-\w]+)$'	This matches a string that has `book/` at the start of the line (**^book/**), then has one or more characters that are *either* a '-' or a word character (**[-\w]+**), and then ends. It also captures this set of characters and passes them to the view in a parameter named 'stub'. This is a fairly typical pattern for a "stub". Stubs are URL-friendly word-based primary keys for data. You might use a stub if you wanted your book URL to be more informative. For example `/catalog/book/the-secret-garden` rather than `/catalog/book/33`.

We can capture multiple patterns in the one match, and hence encode lots of different information in a URL.

Passing additional options in your URL maps

We can declare and pass additional options to the view. The options are declared as a dictionary that we pass as the third un-named argument to the `path()` function. This approach can be useful if we want to use the same view for multiple resources, and pass data to configure its behaviour 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 code into the bottom of the file:

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

We 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 we can access the list of books with the template variable named `object` OR `book` (i.e. generically "*`the_model_name`*").

We can change the template used and the name of the context object used to reference the book in the template. We 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 automatically — in production, this will automatically display an appropriate "resource not found" page, which we have to customise.

The code fragment demonstrates how we would implement the class-based view as a function if we were not using the generic class-based detail view.

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

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

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 to call `render()` with the template name and the book data in the context parameter.

Alternatively, we can use the `get_object_or_404()` function as a shortcut to to raise an `Http404` exception if the record is not found.

```
from django.shortcuts import get_object_or_404

def book_detail_view(request, primary_key):
    book = get_object_or_404(Book, pk=primary_key)
    return render(
        request,
        'catalog/book_detail.html',
        context={'book': book})
```

Creating the Detail View template

Create the HTML file **/locallibrary/catalog/templates/catalog/book_detail.html** with the content. 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
  {% endfor %}
```

- 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 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 we declare a `ForeignKey` (one-to many) field in only the "one" side of the relationship.

Since we don't do anything to declare the relationship in the other ("many") models, it doesn't have any field to get the set of associated records.

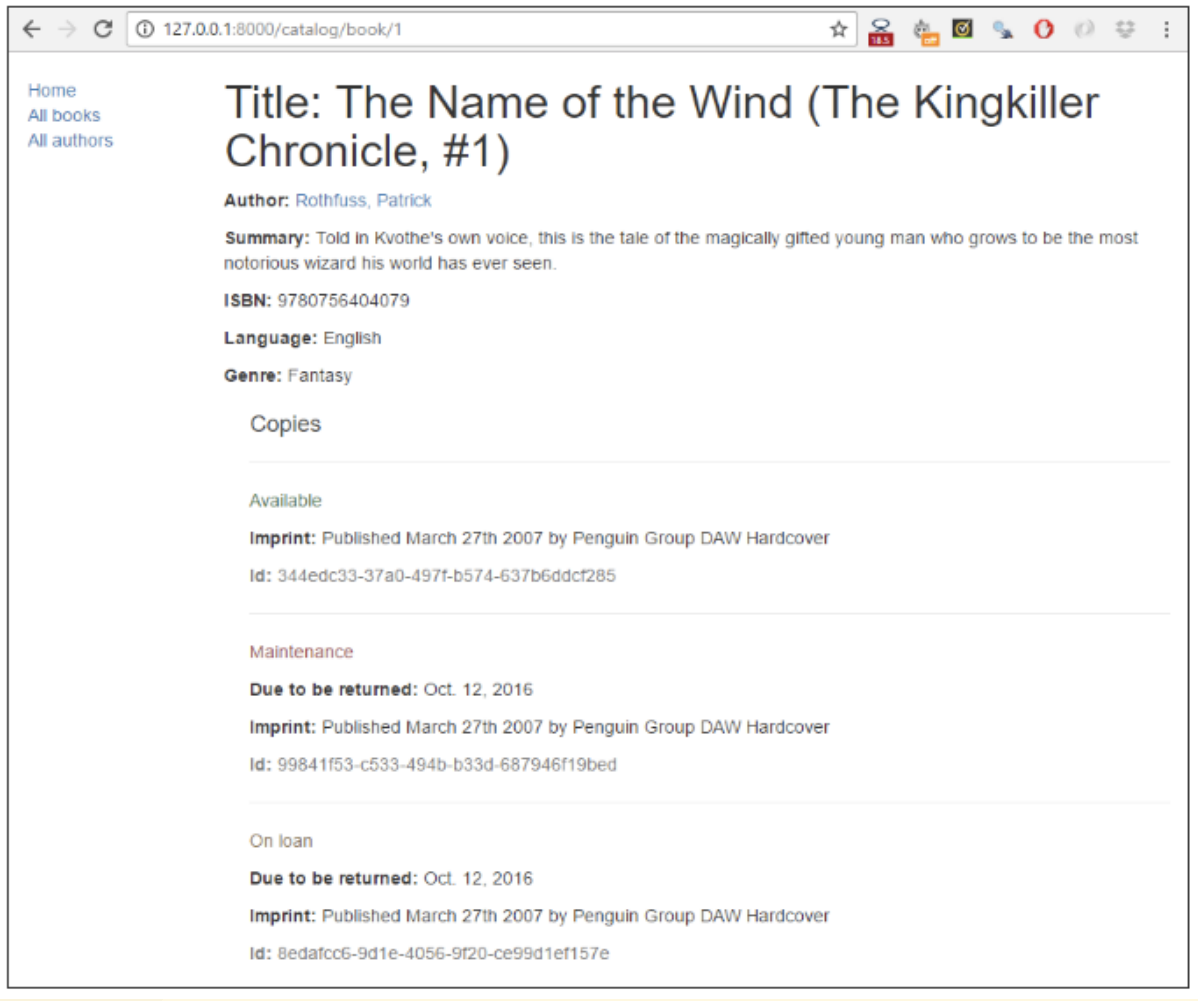
Django constructs an appropriately named "reverse lookup" function that we can use. The name of the function is constructed by lower-casing the model name where the `ForeignKey` was declared, followed by `_set` (i.e. so the function created in `Book` is `bookinstance_set()`).

What does it look like?

We should have created everything needed to display both the book list and book detail pages. Run the server 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 creenshot.



Pagination

If we 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 our list views, reducing the number of items displayed on each page.

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

Views

Open **catalog/views.py**, and add the paginate_by line.

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

With this addition, as soon as we 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 pagination block below our content block. 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 us 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.

What does it look like?

The screenshot shows what the pagination looks like, then we can test it more easily by lowering the number specified in the `paginate_by` line in your **catalog/views.py** file. To get the result we changed it to `paginate_by = 2`.

The pagination links are displayed on the bottom, with next/previous links.

Challenge yourself

The challenge is to create the author detail and list views required to complete the project. These should be made available at the following URLs:

- `catalog/authors/` — The list of all authors.
- `catalog/author/<id>` — The detail view for the specific author with a primary key field named `<id>`

The code required for the URL mappers and the views should be virtually identical to the `Book` list and detail views we created above. The templates will be different but will share similar behaviour.

Your pages should look something like the screenshots.