Tutorial Django Parte 6: Lista genérica e detail views

Este tutorial estende nosso website LocalLibrary, adicionando páginas de lista e detalhes para livros e autores. Aqui, aprenderemos sobre visualizações genéricas baseadas em classe e mostraremos como elas podem reduzir a quantidade de código que você precisa escrever para casos de uso comuns. Também abordaremos o tratamento de URLs em mais detalhes, mostrando como executar a correspondência básica de padrões.

Pré-requisitos:	Conclua todos os tópicos do tutorial anterior, incluindo Django Tutorial Part 5: Creating our home page.
Objetivo:	Para entender onde e como usar modos de exibição genéricos baseados em classe e como extrair padrões de URLs e passar as informações para modos de exibição.

Visão global

Neste tutorial, vamos concluir a primeira versão do website LocalLibrary adicionando páginas de lista e detalhes de livros e autores (ou, para ser mais preciso, mostraremos como implementar as páginas do livro e você mesmo irá criar as páginas dos autores!)

O processo é semelhante à criação da página index, que mostramos no tutorial anterior. Ainda precisamos criar mapas de URL, views e templates. A principal diferença é que, para as páginas de detalhes, teremos o desafio adicional de extrair informações de padrões no URL e passá-las para a visualização. Para essas páginas, demonstraremos um tipo de exibição completamente diferente: lista genérica baseada em classe e exibições detalhadas. Isso pode reduzir significativamente a quantidade de código de visualização necessária, facilitando a gravação e a manutenção.

A parte final do tutorial demonstrará como paginar seus dados ao usar visualizações de lista genéricas baseadas em classe.

Book list page

A página da lista de livros exibirá uma lista de todos os registros de livros disponíveis na página, acessados usando o URL: `catalog/books/` . A página exibirá um título e um autor para cada registro, com o título sendo um hiperlink para a página de detalhes do livro associada. A página terá a mesma estrutura e navegação que todas as outras páginas do site e, portanto, podemos estender o modelo base (**base_generic.html**) que criamos no tutorial anterior.

URL mapping

Abra **/catalog/urls.py** e copie na linha mostrada em negrito abaixo. Quanto à página index, a função `path()` define um padrão para corresponder ao URL (**'books/'**), a função view que será chamado se o URL corresponder (`views.BookListView.as_view()`), e um nome para esse mapeamento específico.

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

Conforme discutido no tutorial anterior, o URL já deve ter correspondencia `/catalog` , então a visualização será realmente chamada para o URL: `/catalog/books/` .

A função view tem um formato diferente do que antes - é porque essa view será realmente implementada como uma classe. Herdaremos de uma função view genérica existente que já faz a maior parte do que queremos que essa função view faça, em vez de escrever a nossa a partir do zero.

Para as class-based views do Django, acessamos uma função de visualização apropriada chamando o método de classe `as_view()` . Isso faz todo o trabalho de criar uma instância da classe e garantir que os métodos do manipulador certo sejam chamados para solicitações HTTP recebidas.

View (class-based)

Poderíamos escrever com facilidade a view da lista de livros como uma função regular (assim como a view index anterior), que consultaria todos os livros no banco de dados e depois chamaria `render()` para passar a lista para um modelo especificado. No entanto, usaremos uma view de

lista genérica class-based (`ListView`) — uma classe que herda de uma view existente. Como a view genérica já implementa a maioria das funcionalidades necessárias e segue as práticas recomendadas do Django, poderemos criar uma exibição de lista mais robusta com menos código, menos repetições e, finalmente, menos manutenção.

Abra **catalog/views.py**, e copie o seguinte código na parte inferior do arquivo:

```
from django.views import generic

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

É isso aí! A view genérica consultará o banco de dados para obter todos os registros para o modelo especificado (`Book`) em seguida, renderize um template localizado em **/locallibrary/catalog/templates/catalog/book_list.html** (que criaremos abaixo). Dentro do template, você pode acessar a lista de livros com a variável de template denominada `object_list` OU `book_list` (i.e. genericamente "*`the_model_name`*`_list`").

> **Nota**: Esse caminho estranho para a localização do template não é um erro de impressão - as visualizações genéricas procuram modelos em *`/application_name/the_model_name`*`_list.html` (`catalog/book_list.html` nesse caso) dentro do aplicativo *`/application_name/`*`templates/` diretório (`/catalog/templates/`).

Você pode adicionar atributos para alterar o comportamento padrão acima. Por exemplo, você pode especificar outro arquivo do template se precisar ter várias visualizações que usem esse mesmo modelo ou se desejar usar um nome de variável de template diferente se `book_list` não é intuitivo para o seu caso de uso de template específico. Possivelmente, a variação mais útil é alterar/filtrar o subconjunto de resultados retornados - portanto, em vez de listar todos os livros, você pode listar os cinco principais livros que foram lidos por outros usuários.

```
class BookListView(generic.ListView):
    model = Book
    context_object_name = 'my_book_list'   # your own name for the list
    queryset = Book.objects.filter(title__icontains='war')[:5] # Get 5 b
    template_name = 'books/my_arbitrary_template_name_list.html'  # Spec
```

Substituindo métodos em class-based views

Embora não precisemos fazer isso aqui, você também pode substituir alguns dos métodos da classe.

Por exemplo, podemos substituir o método `get_queryset()` para alterar a lista de registros retornados. Isso é mais flexível do que apenas definir o atributo `queryset` como fizemos no fragmento de código anterior (embora não haja nenhum benefício real neste caso):

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

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

Também podemos substituir `get_context_data()` para passar variáveis de contexto adicionais para o template (por exemplo, a lista de livros é passada por padrão). O fragmento abaixo mostra como adicionar uma variável chamada "`some_data`" para o contexto (estaria disponível como uma variável de template).

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

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get the context
        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
```

Ao fazer isso, é importante seguir o padrão usado acima:

- Primeiro obtenha o contexto existente da nossa superclasse.
- Em seguida, adicione as novas informações de contexto

- Em seguida, adicione as novas informações de contexto.
- Em seguida, retorne o novo contexto (atualizado).

> **Nota**: Confira Built-in class-based generic views (Django docs) para muitos mais exemplos do que você pode fazer.

Criando o template List View

Crie o arquivo HTML **/locallibrary/catalog/templates/catalog/book_list.html** e copie o texto abaixo. Como discutido acima, este é o arquivo de template padrão esperado pela list view genérica da class-based view (para um modelo chamado `Book` em um aplicativo chamado `catalog`).

Os templates para visualizações genéricas são como qualquer outro template (*embora*, é claro, o contexto/informações passadas para o template possam ser diferentes). Assim como em nosso template *index*, estendemos nosso template base na primeira linha e substituímos o bloco denominado `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> ({{bc
      </li>
    {% endfor %}
  </ul>
  {% else %}
    <p>There are no books in the library.</p>
  {% endif %}
{% endblock %}
```

A view passa o contexto (lista de livros), por padrão, um `object_list` e `book_list` aliases;

qualquer um funcionará.

Execução conditional

Nós usamos o `if`, `else`, e `endif` template tags para verificar se o `book_list` foi definido e não está vazio. E se `book_list` está vazio, então a cláusula `else` exibe o texto explicando que não há livros para listar. E se `book_list` não estiver vazio, percorreremos a lista de livros.

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

A condição acima verifica apenas um caso, mas você pode testar em condições adicionais usando a template tag `elif` (e.g. `{% elif var2 %}`). Para obter mais informações sobre operadores condicionais, consulte: if, ifequal/ifnotequal, e ifchanged em Built-in template tags and filters (Django Docs).

For loops

O template usa as template tags for e `endfor` para percorrer a lista de livros, como mostrado abaixo. Cada iteração preenche a variável de template `book` com informações para o item da lista atual.

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

Embora não seja usado aqui, dentro do loop, o Django também criará outras variáveis que você pode usar para rastrear a iteração. Por exemplo, você pode testar a variável `forloop.last` para executar o processamento condicional na última vez em que o loop é executado.

Acessando variáveis

O código dentro do loop cria um item de lista para cada livro que mostra o título (como um link para a exibição de detalhes ainda a ser criada) e o autor.

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

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

Acessamos os *campos* do registro de livro associado usando a "notação de ponto" (e.g. `book.title e book.author`), onde o texto após o item `book` é o nome do campo (conforme definido no modelo).

Também podemos chamar *funções* no modelo de dentro do nosso template - nesse caso, chamamos `Book.get_absolute_url()` para obter um URL que você pode usar para exibir o registro de detalhe associado. Isso funciona desde que a função não tenha argumentos (não há como passar argumentos!)

> **Nota**: Temos que ter um pouco de cuidado com os "efeitos colaterais" ao chamar funções em templates. Aqui apenas exibimos um URL, mas uma função pode fazer praticamente qualquer coisa - não queremos excluir nosso banco de dados (por exemplo) apenas renderizando nosso template!

Atualize o template base

Abra o template base (**/locallibrary/catalog/templates/*base_generic.html***) e insira **{% url 'books' %}** no link da URL para **All books**,como mostrado abaixo. Isso habilitará o link em todas as páginas (podemos colocá-lo em prática agora que criamos o mapeador de URL "books").

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

Com o que se parece?

Ainda não será possível criar a lista de livros, porque ainda falta uma dependência - o mapa de URL para as páginas de detalhes do livro, necessário para criar hiperlinks para livros individuais. Mostraremos as visualizações de lista e de detalhes após a próxima seção.

Pagina Book detail

A página book detail exibirá informações sobre um livro específico, acessado usando o URL `catalog/book/<id>` (onde `<id>` é a chave primária do livro). Além dos campos no model `Book` (author, summary, ISBN, language, e genre), também listaremos os detalhes das cópias

Book (author, summary, ISBN, language, e genre), também listaremos os detalhes das cópias disponíveis (`BookInstances`) incluindo o status, data prevista de retorno, impressão e ID. Isso permitirá que nossos leitores não apenas saibam sobre o livro, mas também confirmem se/quando ele está disponível.

URL mapping

Abra **/catalog/urls.py** e adicione a URL '**book-detail**' mostrado em negrito abaixo. Esta função `path()` define um padrão, exibição de detalhes genérica associada à classe associada e um nome.

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

Para o path *book-detail* o padrão de URL usa uma sintaxe especial para capturar o ID específico do livro que queremos ver. A sintaxe é muito simples: colchetes angulares definem a parte da URL a ser capturada, incluindo o nome da variável que a view pode usar para acessar os dados capturados. Por exemplo, **<something>** , capturará o padrão marcado e passará o valor para a visualização como uma variável "alguma coisa". Opcionalmente, você pode preceder o nome da variável com um converter specification que define o tipo de dados (int, str, slug, uuid, path).

Neste caso, usamos `'<int:pk>'` para capturar o ID do livro, que deve ser uma sequência especialmente formatada e passá-la para a view como um parâmetro chamado `pk` (abreviatura de primary key). Esta é a id que está sendo usado para armazenar o livro exclusivamente no banco de dados, conforme definido no Book Model.

> **Nota**: Como discutido anteriormente, nosso URL correspondente é realmente `catalog/book/<digits>` (porque estamos no aplicativo de **catalog**, `/catalog/` é assumido).

> **Importante**: A view de detalhes genérica class-based *espera* receber um parâmetro chamado **pk**. Se você estiver escrevendo sua própria função view, poderá usar o nome de qualquer parâmetro que desejar, ou mesmo transmitir as informações em um argumento sem

qualquer parâmetro que desejar, ou mesmo transmitir as informações em um argumento sem nome.

Correspondência avançada de caminhos/iniciador de expressão regular

> **Nota**: Você não precisará desta seção para concluir o tutorial! Nós fornecemos isso porque conhecer essa opção provavelmente será útil no seu futuro centrado no Django.

The pattern matching provided by `path()` is simple and useful for the (very common) cases where you just want to capture *any* string or integer. If you need more refined filtering (for example, to filter only strings that have a certain number of characters) then you 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. For example, the previous path could have been written as shown below:

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

Regular expressions are an incredibly powerful pattern mapping tool. They are, frankly, quite unintuitive and scary for beginners. Below is a very short primer!

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>'**).

The main parts of the syntax you will need to know for declaring the pattern matches are:

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 (_)

Symbol	Meaning
+	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.

Most other characters can be taken literally!

Let's consider a few real examples of patterns:

Pattern	Description
	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).
r'^book/(?P<pk>\d+)$'	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.

Pattern	Description
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`.

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

> **Nota**: Como desafio, considere como você pode codificar um URL para listar todos os livros lançados em um determinado ano, mês, dia e o RE que poderia ser usado para correspondê-lo.

Passando opções adicionais em seus mapas de URL

Um recurso que não usamos aqui, mas que você pode achar valioso, é que você pode declarar e passar opções adicionais para a view. As opções são declaradas como um dicionário que você passa como o terceiro argumento sem nome para a função `path()`. Essa abordagem pode ser útil se você desejar usar a mesma visualização para vários recursos e transmitir dados para configurar seu comportamento em cada caso (abaixo, fornecemos um modelo diferente em cada caso).

```
path('url/', views.my_reused_view, {'my_template_name': 'some_path'}, na
path('anotherurl/', views.my_reused_view, {'my_template_name': 'another_
```

> **Nota:** As opções extras e os padrões capturados nomeados são passados para a view como argumentos *nomeados*. Se você usar o **mesmo nome** para um padrão capturado e uma opção extra, somente o valor do padrão capturado será enviado para a visualização (o valor especificado na opção adicional será descartado).

View (class-based)

Abra **catalog/views.py**, e copie o seguinte código na parte inferior do arquivo:

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

É isso aí! Tudo o que você precisa fazer agora é criar um modelo chamado **/locallibrary/catalog/templates/catalog/book_detail.html**, e a visualização passará as informações do banco de dados para o registro `Book` extraído pelo mapeador de URL. Dentro do modelo, você pode acessar a lista de livros com a variável de modelo denominada `object` ou `book` (i.e. genericamente "*`the_model_name`*").

Se necessário, você pode alterar o template usado e o nome do objeto de contexto usado para referenciar o livro no template. Você também pode substituir métodos para, por exemplo, adicionar informações adicionais ao contexto.

O que acontece se o registro não existir?

Se um registro solicitado não existir, a view de detalhes genérica class-based levantará uma exceção `Http404` para você automaticamente — em produção, isso exibirá automaticamente uma página apropriada de "resource not found", que você pode personalizar se desejar.

Apenas para lhe dar uma idéia de como isso funciona, o fragmento de código abaixo demonstra como você implementaria a exibição baseada em classe como uma função se você não estivesse usando a view de detalhe genérica class-based.

```
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':
```

A view tenta primeiro obter o registro de livro específico do modelo. Se isso falhar, a view deve gerar uma exceção `Http404` para indicar que o livro "não foi encontrado". A etapa final é, como sempre, chamar `render()` com o nome do template e os dados do livro no parâmetro `context` (como um dicionário).

Como alternativa, podemos usar a função `get_object_or_404()` como um atalho para levantar uma exceção `Http404` se o registro não for encontrado.

```
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':
```

Criando o template Detail View

Crie o arquivo HTML **/locallibrary/catalog/templates/catalog/book_detail.html** e forneça o conteúdo abaixo. Conforme discutido acima, este é o nome do arquivo de template padrão esperado pela view de *detalhes* genérica class-based (para um modelo chamado `Book` no aplicativo chamado `catalog`).

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

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

  <p><strong>Author:</strong> <a href="">{{ book.author }}</a></p> <!--
  <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

  <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.stat
      {% 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 %}
```

> O link do autor no template acima tem um URL vazio porque ainda não criamos uma página de detalhes do autor. Uma vez que isso exista, você deve atualizar o URL assim:
>
> ```
> <a href="{% url 'author-detail' book.author.pk %}">{{ book.author }}</a>
> ```

Embora um pouco maior, quase tudo neste template foi descrito anteriormente:

- Estendemos nosso modelo básico e substituímos o bloco "content".
- Usamos o processamento condicional para determinar se deve ou não exibir conteúdo específico.
- Usamos `for` loops para percorrer as listas de objetos.
- Acessamos os campos de contexto usando a notação de ponto (porque usamos a exibição genérica detalhada, o contexto é chamado `book`; também poderíamos usar "`object`")

A única coisa interessante que não vimos antes é a função `book.bookinstance_set.all()`. Este método é "automagicamente" construído pelo Django para retornar o conjunto de registros `BookInstance` associados com um `Book` em particular.

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

Este método é necessário porque você declara um campo `ForeignKey` (um-para-muitos) somente no lado "um" do relacionamento. Como você não faz nada para declarar o

relacionamento nos outros modelos ("muitos"), ele não possui nenhum campo para obter o conjunto de registros associados. Para superar esse problema, o Django constrói uma função "pesquisa reversa" chamada de forma apropriada, que você pode usar. O nome da função é

construído com letras minúsculas no nome do modelo em que o `ForeignKey` foi declarado, seguido por `_set` (i.e. então a função criada em `Book` é `bookinstance_set()`).

> **Nota**: Aqui usamos `all()` para obter todos os registros (o padrão). Enquanto você pode usar o método `filter()` para obter um subconjunto de registros no código, não é possível fazer isso diretamente nos modelos, porque não é possível especificar argumentos para funções.
>
> Observe também que, se você não definir um pedido (na sua class-based view ou modelo), também verá erros do servidor de desenvolvimento como este:
>
> ```
> [29/May/2017 18:37:53] "GET /catalog/books/?page=1 HTTP/1.1" 200 1637
> /foo/local_library/venv/lib/python3.5/site-packages/django/views/generic/list.py:99:
>   allow_empty_first_page=allow_empty_first_page, **kwargs)
> ```
>
> Isso acontece porque o objeto paginator espera ver algum ORDER BY sendo executado no seu banco de dados subjacente. Sem ele, não é possível garantir que os registros que estão sendo retornados estejam na ordem certa!
>
> Este tutorial não atingiu a **Paginação** (ainda, mas em breve), mas como você não pode usar `sort_by()` e passar um parâmetro (o mesmo com `filter()` descrito acima), você terá que escolher entre três opções:
>
> 1. Adicione um `ordering` dentro de uma declaração `class Meta` no seu model.
> 2. Adicione um atributo `queryset` na sua class-based view, especificando um `order_by()`.
> 3. Adicione um método `get_queryset` à sua class-based view personalisada e também especifique o `order_by()`.
>
> Se você decidir ir com uma `class Meta` no model `Author` (provavelmente não tão flexível quanto personalizar o class-based view, mas fácil o suficiente), você terminará com algo assim:
>
> ```
> class Author(models.Model):
> ```

```
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    date_of_birth = models.DateField(null=True, blank=True)
    date_of_death = models.DateField('Died', null=True, blank=True)

    def get_absolute_url(self):

        return reverse('author-detail', args=[str(self.id)])

    def __str__(self):
        return f'{self.last_name}, {self.first_name}'

    class Meta:
        ordering = ['last_name']
```

Obviamente, o campo não precisa ser `last_name` : poderia ser qualquer outro.

E por último, mas não menos importante, você deve classificar por um atributo/coluna que realmente tenha um índice (exclusivo ou não) em seu banco de dados para evitar problemas de desempenho. Obviamente, isso não será necessário aqui (e provavelmente estamos nos adiantando muito) com tão poucos livros (e usuários!), Mas é algo a ser lembrado em projetos futuros.

Com o que se parece agora?

Nesse ponto, deveríamos ter criado tudo o necessário para exibir a lista de livros e as páginas de detalhes do livro. Execute o servidor (`python3 manage.py runserver`) e abra no seu navegador http://127.0.0.1:8000/.

> **Aviso:** Não clique em nenhum autor ou link de detalhes do autor ainda - você os criará no desafio!

Clique no link **All books** para exibir a lista de livros.

.

Em seguida, clique no link de um dos seus livros. Se tudo estiver configurado corretamente, você deverá ver algo como a seguinte captura de tela.

.

Paginação

Se você tiver apenas alguns registros, nossa página da lista de livros ficará bem. No entanto, à medida que você entra nas dezenas ou centenas de registros, a página levará progressivamente mais tempo para carregar (e terá muito conteúdo para navegar com sensatez). A solução para esse problema é adicionar paginação às exibições de lista, reduzindo o número de itens exibidos em cada página.

O Django possui excelente suporte embutido para paginação. Melhor ainda, isso é incorporado às exibições de lista genéricas baseadas em classes, para que você não precise fazer muito para habilitá-lo!

Views

Abra **catalog/views.py**, e adicionea linha `paginate_by` mostrado em negrito abaixo.

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

Com essa adição, assim que você tiver mais de 10 registros, a visualização começará a paginar os dados que envia para o modelo. As diferentes páginas são acessadas usando os parâmetros GET - para acessar a página 2, você usaria o URL: `/catalog/books/`**`?page=2`** .

Templates

Agora que os dados estão paginados, precisamos adicionar suporte ao modelo para rolar pelo conjunto de resultados. Como podemos fazer isso em todas as visualizações de lista, faremos isso de uma maneira que possa ser adicionada ao modelo base.

Abra **/locallibrary/catalog/templates/*base_generic.html*** e copie no seguinte bloco de paginação, abaixo do nosso bloco de conteúdo (destacado abaixo em negrito). O código primeiro verifica se a paginação está ativada na página atual. Nesse caso, adiciona os links seguintes e anteriores, conforme apropriado (e o número da página atual).

```
{% 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.previou
                {% endif %}
                <span class="page-current">
                    Page {{ page_obj.number }} of {{ page_obj.paginator.
                </span>
                {% if page_obj.has_next %}
                    <a href="{{ request.path }}?page={{ page_obj.next_pa
                {% endif %}
            </span>
        </div>
    {% endif %}
{% endblock %}
```

O `page_obj` é um objeto de Paginator que existirá se a paginação estiver sendo usada na página atual. Permite obter todas as informações sobre a página atual, as páginas anteriores, quantas páginas existem, etc.

Usamos `{{ request.path }}` para obter o URL da página atual para criar os links de paginação. Isso é útil porque é independente do objeto que estamos paginando.

É isso aí!

Com o que se parece agora?

A captura de tela abaixo mostra a aparência da paginação - se você não inseriu mais de 10 títulos no banco de dados, pode testá-lo com mais facilidade, abaixando o número especificado na linha `paginate_by` no seu arquivo **catalog/views.py**. Para obter o resultado abaixo, alteramos para `paginate_by = 2`.

Os links de paginação são exibidos na parte inferior, com os links seguinte/anterior, dependendo da página em que você está.

.

Challenge yourself

The challenge in this article 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.

> **Note**:
>
> - Once you've created the URL mapper for the author list page you will also need to update the **All authors** link in the base template. Follow the same process as we did when we updated the **All books** link.
> - Once you've created the URL mapper for the author detail page, you should also update the book detail view template (**/locallibrary/catalog/templates/catalog/book_detail.html**) so that the author link points to your new author detail page (rather than being an empty URL). The line will change to add the template tag shown in bold below.
>
>   ```
>   <p>Author: <a href="{% url 'author-detail' book
>   ```

When you are finished, your pages should look something like the screenshots below.

.

.

Summary

Congratulations, our basic library functionality is now complete!

In this article, we've learned how to use the generic class-based list and detail views and used them to create pages to view our books and authors. Along the way we've learned about pattern matching with regular expressions, and how you can pass data from URLs to your views. We've also learned a few more tricks for using templates. Last of all we've shown how to paginate list

also learned a few more tricks for using templates. Last of all we've shown how to paginate list views so that our lists are manageable even when we have many records.

In our next articles, we'll extend this library to support user accounts, and thereby demonstrate user authentication, permissons, sessions, and forms.

See also

- Built-in class-based generic views (Django docs)
- Generic display views (Django docs)
- Introduction to class-based views (Django docs)
- Built-in template tags and filters (Django docs).
- Pagination (Django docs)

Neste módulo

Last modified: 26 de ago. de 2020, by MDN contributors

Change your language

Português (do Brasil) Change language