**Website de uma biblioteca.**

**Onde parei:** [Django Tutorial Part 8: User authentication and permissions - Learn web development | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication)

Para acessar a página do site depois de pronto 127.0.0.1:8000/admin:([Administração do Site | Site de administração do Django](http://127.0.0.1:8000/admin/))

Pagina do site: <http://127.0.0.1:8000/>.

Pagina de login: http://127.0.0.1:8000/accounts/

Local do arquivo na máquina: C:\Users\Claudio\OneDrive\Univesp\LocalLibrary\locallibrary\locallibrary.

Esquema básico da biblioteca a ser construída conforme a seguinte página: [Tutorial Django: Website da Biblioteca Local - Aprendendo desenvolvimento web | MDN (mozilla.org)](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Tutorial_local_library_website) acessado em 20/05/2022 as 20:41 hs.

Diagrama

Descrição gerada automaticamente

Tutorial para o Windows.

Baixar e instalar o Python: [Download Python | Python.org](https://www.python.org/downloads/).

Criar um ambiente virtual de trabalho.

Abrir o prompt de comando (cmd) e digitar: pip install virtualenvwrapper-win. (só uma vez)

Para criar um ambiente virtual digitar a seguir mkdirvirtualenv (nome do ambiente).

Existem apenas alguns poucos comandos que você deveria conhecer (há mais comandos que você pode encontrar na documentação da ferramenta, porém, os comandos abaixo serão os que você usará regularmente):

* deactivate — Encerra o ambiente virtual Python corrente.
* workon — Lista ambientes virtuais disponíveis.
* workon name\_of\_environment — Ativa o ambiente virtual Python especificado.
* rmvirtualenv name\_of\_environment — Remove o ambiente especificado.

Instale o Django no ambiente virtual com o comando: pip3 install django.

Dentro do prompt de comando navegue até a pasta onde será criado o projeto e crie a pasta comando: mikdir (nomedodiretório-pasta).

Entre na pasta (cd nomedodiretório-pasta) e digite: django-admin startproject (nomedosite).

[**Criando o projeto**](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/skeleton_website#criando_o_projeto)

Primeiro abra o prompt de comando/terminal (tenha certeza que está em seu [ambiente virtual)](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/development_environment), navegue até o diretório que deseja colocar seus aplicativos Django (coloque em um lugar fácil de achar, como dentro da pasta *documentos*), e crie uma pasta para seu novo website (nesse caso: *nomedodiretorio-pasta*). Acesse então a pasta usando o comando cd:

mkdir locallibrary

cd locallibrary

Copy to Clipboard

Crie um novo projeto usando o comando django-admin startproject, como mostrado abaixo, e entre nessa pasta.

django-admin startproject locallibrary

cd locallibrary

Copy to Clipboard

O comando django-admin cria uma estrutura com pastas e arquivos como a mostrada abaixo:

locallibrary/

manage.py

locallibrary/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

Copy to Clipboard

Nosso diretório de trabalho atual deve parecer com isso:

../nomedodiretorio-pasta/locallibrary/

  A sub-pasta do projeto *locallibrary* será a raíz para nosso site:

* **\_\_init\_\_.py**é um arquivo em branco que instrui o Python a tratar esse diretório como um pacote Python.
* **settings.py** contém todas as definições do website. É onde nós registramos qualquer aplicação que criarmos, a localização de nossos arquivos estáticos, configurações de banco de dados etc.
* **urls.py** define os mapeamentos de URL para visualização do site. Mesmo que esse arquivo possa conter *todo*o código para mapeamento de URL, é mais comum delegar apenas o mapeamento para aplicativos específicos, como será visto mais adiante.
* **wsgi.py** é usado para ajudar na comunicação entre seu aplicativo Django e o web server. Você pode tratar isso como um boilerplate.

O script **manage.py** é usado para criar aplicações, trabalhar com bancos de dados, e iniciar o webserver de desenvolvimento.

[**Criando o aplicativo de catálogo**](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/skeleton_website#criando_o_aplicativo_de_cat%C3%A1logo)

Agora execute o seguinte comando para criar o *catálogo* da aplicação que fará parte de nosso projeto localibrary (o comando deve ser executado na mesma pasta que está o **manage.py** do seu projeto):

py3 manage.py startapp catalog

O diretório do projeto atualizado deve parecer com esse:

locallibrary/

manage.py

locallibrary/

catalog/

admin.py

apps.py

models.py

tests.py

views.py

\_\_init\_\_.py

migrations/

Abra o arquivo de configurações do projeto **locallibrary/locallibrary/settings.py** e encontre a definição para a lista INSTALLED\_APPS. Agora adicione uma nova linha no fim da lista, como a mostrada em negrito abaixo. (para acessar o arquivo settings.py digite **nano settings.py** no prompt de comando e aperte enter/ para encerrar control x e yes seguido de enter)

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

**'catalog.apps.CatalogConfig',**

]

## [Especificando o Banco de Dados](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/skeleton_website#especificando_o_banco_de_dados)

Tipicamente, esse é o momento em que você também especifica o  banco de dados que será usado no projeto— faz mais sentido usar o mesmo banco de dados tanto para desenvolvimento quanto para a produção (quando possível), a fim de evitar pequenas diferenças de comportamento. Você pode encontrar mais sobre as outras opções em [Databases](https://docs.djangoproject.com/en/2.0/ref/settings/" \l "databases) (Documentação Django).

Usaremos o banco de dados SQLite para este exemplo porque não esperamos ter muito acesso simultâneo em um banco de dados para demonstração, e também porque ele não requer trabalho adicional de configuração! Você pode ver como o banco de dados é configurado em **settings.py** (mais informações estão incluidas abaixo).

#settings.py

**import os**

from pathlib import Path

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': **os.path.join(BASE\_DIR, 'db.sqlite3'),**

}

}

Copy to Clipboard

Já que nós estamos usando SQLite, nós não precisamos de nenhum outro passo aqui. Vamos ir em frente!

## [Outras configurações do projeto](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/skeleton_website#outras_configura%C3%A7%C3%B5es_do_projeto)

O arquivo **settings.py** também é usado para configurar várias outras definições, mas por ora você provavelmente quer mudar apenas a [TIME\_ZONE](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-TIME_ZONE) — deve se utilizar uma string padrão da [Lista de tz time zones](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones) (a coluna TZ na tabela contém os valores que você precisa). Mude seu valor de TIME\_ZONE para uma string relativa ao seu fuso-horário, por exemplo:

TIME\_ZONE = 'America/Sao\_Paulo'

## [Conectando o mapeador de URL](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/skeleton_website#conectando_o_mapeador_de_url)

O website foi criado com um arquivo mapeador de URL (**urls.py**) na pasta do projeto. Embora você possa usar esse arquivo para gerenciar todos seus mapeamentos de URL, é mais comum fazer os mapeamentos diretamente no aplicativo associado.

Abra **locallibrary/locallibrary/urls.py** e leia o texto que explica alguma formas de usar o mapeador de URL.

"""locallibrary URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:

https://docs.djangoproject.com/en/2.0/topics/http/urls/

Examples:

Function views

1. Add an import: from my\_app import views

2. Add a URL to urlpatterns: path('', views.home, name='home')

Class-based views

1. Add an import: from other\_app.views import Home

2. Add a URL to urlpatterns: path('', Home.as\_view(), name='home')

Including another URLconf

1. Import the include() function: from django.urls import include, path

2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))

"""

from django.contrib import admin

from django.urls import path

urlpatterns = [

path('admin/', admin.site.urls),

]

Copy to Clipboard

Os mapeamentos de URL são gerenciados através da variável urlpatterns  que é uma lista Python de funções path(). Cada função path() associa um padrão de URL para uma view específica, que será exibida quando o padrão for correspondido, ou com outra lista de testes de padrões de URL (no segundo caso, o padrão vem da "URL base" para padrões definidos no módulo target). A lista urlpatterns define inicialmente uma função única que mapeia todas URLs com o padrão admin para o módulo admin.site.urls, que contém as próprias definições de mapeamento de URL da área de administração do aplicativo.

Adicione as linhas abaixo no fim do arquivo a fim de adicionar um novo item à lista urlpatterns. Esse novo item inclui um  path() que encaminha solicitações com o padrão catalog/ para o módulo catalog.urls (o arquivo com a URL relativa **/catalog/urls.py**).

# Use include() to add paths from the catalog application

from django.conf.urls import include

from django.urls import path

urlpatterns += [

path('catalog/', include('catalog.urls')),

]

Copy to Clipboard

Agora iremos mudar a URL raíz de nosso site (i.e. 127.0.0.1:8000) para 127.0.0.1:8000/catalog/; pois esse é o único app que iremos usar neste projeto. Para isso, usaremos uma função view especial (RedirectView), que leva como primeiro argumento a nova URL relativa para redirecionar para /catalog/ quando o padrão URL especificado na função path() for chamado (a URL raíz nesse caso).

Adicione as linhas abaixo, novamente no fim do arquivo:

#Add URL maps to redirect the base URL to our application

from django.views.generic import RedirectView

urlpatterns += [

path('', RedirectView.as\_view(url='/catalog/')),

]

Copy to Clipboard

Deixe o primeiro parâmetro da função path vazio, implicando em '/'. Se você escrever o primeiro parâmetro como '/', Django irá te mostar o seguinte aviso assim que iniciar o servidor de desenvolvimento.

System check identified some issues:

WARNINGS:

?: (urls.W002) Your URL pattern '/' has a route beginning with a '/'.

Remove this slash as it is unnecessary.

If this pattern is targeted in an include(), ensure the include() pattern has a trailing '/'.

Copy to Clipboard

Por padrão, Django não "serve" arquivos estáticos como CSS, JavaScript e imagens, mas ele pode ser útil para o servidor web de desenvolvimento enquanto você cria seu site. Como comentário final sobre o mapeador de URL, você pode habilitar a veiculação de arquivos estáticos durante o desenvolvimento adicionando as seguintes linhas.

Coloque o seguinte bloco no fim do arquivo:

# Use static() to add url mapping to serve static files during development (only)

from django.conf import settings

from django.conf.urls.static import static

urlpatterns += static(settings.STATIC\_URL, document\_root=settings.STATIC\_ROOT)

**Retorne ao diretório onde você encontra o manage.py:**

locallibrary/

manage.py

locallibrary/

catalog/

**Execute os comandos: py mange.py makemigrations (enter)**

**py manage.py migrate (enter)**

## [Projetando os modelos para o website](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#projetando_os_modelos_para_o_website_locallibrary)[LocalLibrary](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial_local_library_website)

Antes de começarmos a criar nossos modelos, vale a pena perder alguns minutos pensando sobre os dados que iremos guardar e as relações entre os diferentes modelos que serão criados.

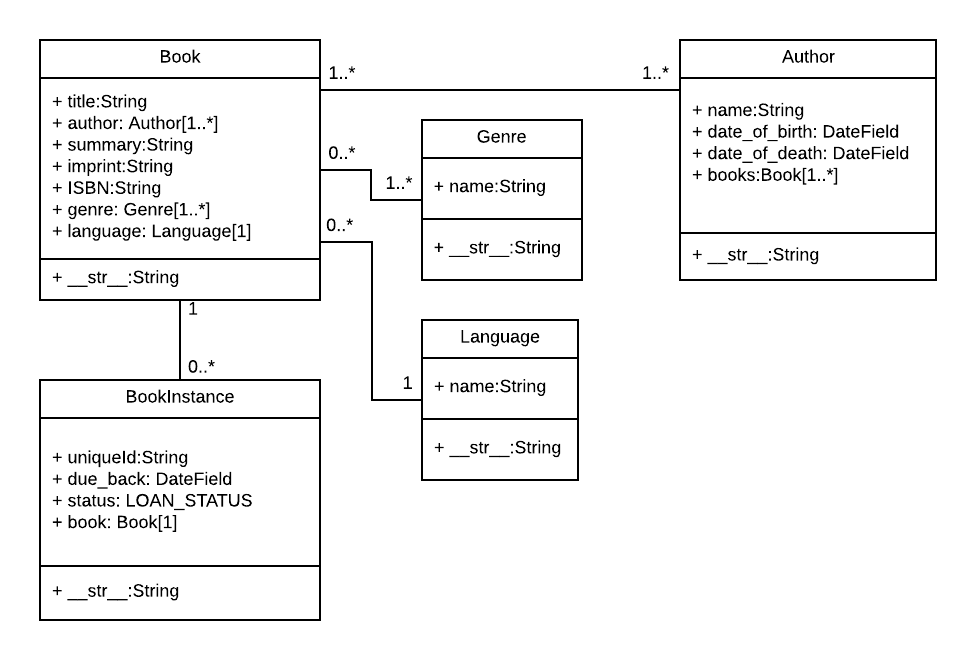
Sabemos que precisamos armazenar informação sobre os livros (título, resumo, autor, idioma, gênero, ISBN) e também que existem várias cópias do mesmo livro na biblioteca (com um id único, status de disponibilidade, etc.). Talvez queiramos armazenar mais informações sobre o autor além de somente seu nome, até porque existem vários autores com o mesmo nome, ou com nomes parecidos. Queremos ordernar a busca dos livros por título, autor, idioma e gênero.

Quando estamos projetando nossos modelos, faz sentido criar modelos separados para cada "objeto". Em nosso caso de estudo, os "objetos" são os livros (a informação de cada livro, não a cópia em si), as cópias dos livros (um livro pode ter mais de uma cópia) e os autores.

Você pode também utilizar modelos para representar opções em uma lista de seleção (por exemplo numa lista suspensa), o que é melhor do que trabalhar com opções predefinidas — isso é recomendado quando nem todas as opções são conhecidas ou podem mudar de acordo com um filtro. Obviamente, para nosso tutorial, modelos candidatos para esse caso são o gênero e o idioma.

Após decidirmos nossos modelos e campos, precisamos pensar no relacionamento dessas informações. Django permite que você defina relações que são um pra um (OneToOneField), um pra muitos (ForeignKey) e muitos pra muitos (ManyToManyField).

Com isso em mente, os diagramas UML de associação, mostram abaixo os modelos que definiremos nesse caso (como caixas).



Como acima, criamos modelos para Book (que contém os detalhes genéricos do livro),  
BookInstance (contém os status das cópias físicas e específicas dos livros disponíveis no sistema) e Author. Também decidimos ter um modelo para o gênero (Genre), para que os valores possam ser criados/selecionados através da interface administrativa. Decidimos não ter um modelo para o BookInstance: status - pois, codificamos os valores em (LOAN\_STATUS) porque não esperamos que isso mude. Dentro de cada uma das caixas você pode ver o nome do modelo, os campos nomes e tipos e também os métodos e seus tipos de retorno.

O diagrama também mostra as relações entre os modelos, incluindo suas multiplicidades. As multiplicidades são os números no diagrama que mostram as quantidades (máxima e mínima) que cada modelo pode estar presente nos relacionamentos. Por exemplo, a linha que conecta as caixas mostra que Book e um Genre estão relacionados. Os números próximos ao modelo Genre mostram que um livro deve ter um ou mais gêneros (ou quantos você quiser), enquanto os números do outro lado da linha, ao lado do modelo Book mostram que um gênero pode ter zero ou muitos livros associados.

**Nota**: A próxima seção fornece uma explicação básica sobre como os modelos são definidos e usados. Ao ler sobre isso, considere como vamos construir cada um dos modelos conforme o diagrama acima.

## [Model primer](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#model_primer)

Esta seção fornece uma breve visão sobre como um modelo é definido e alguns dos mais importantes campos e argumentos dos campos.

### [Definição do Modelo](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#defini%C3%A7%C3%A3o_do_modelo)

Modelos são geralmente definidos no arquivo **models.py** em uma aplicação. Eles são implementados como subclasse de django.db.models.Model, e podem incluir campos, métodos e metadados. O fragmento de código abaixo, mostra um modelo "típico", chamado MyModelName:

from django.db import models

class MyModelName(models.Model):

    """Uma típica classe definindo um modelo, derivada da classe Model."""

  # Campos

    my\_field\_name = models.CharField(max\_length=20, help\_text='Enter field documentation')

  ...

  # Metadados

class Meta:

  ordering = ['-my\_field\_name']

  # Métodos

    def get\_absolute\_url(self):

       """Retorna a url para acessar uma instancia específica de MyModelName."""

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

    def \_\_str\_\_(self):

""" String para representar o objeto MyModelName (no site Admin)."""

        return self.my\_field\_name

Nas seções abaixa, exploraremos detalhadamente cada um dos recursos dentro do modelo:

#### Campos (Fields)

Um modelo pode ter um número árbitrário de campos, de qualquer tipo -- cada um representa uma coluna de dados que queremos armazenar em uma de nossas tabelas de banco de dados. Cada registro do banco de dados (row - linha) consitirá em um valor de cada campo. Vamos ver o exemplo visto acima:

my\_field\_name = models.CharField(max\_length=20, help\_text='Enter field documentation')

Copy to Clipboard

Nosso exemplo acima tem um único campo chamado my\_field\_name, do tipo models.CharField - o que significa que este campo conterá strings de caracteres alfanuméricos. Os tipos de cada campo são atribuídos usando classes específicas, que determinam o tipo de registro usado para armazenar os dados no banco de dados, juntamente com os critérios de validação a serem usados ​​quando os valores são recebidos de um formulário HTML (ou seja, o que constitui um valor válido). Os tipos de cada campo também podem receber argumentos que especifiquem como o campo é armazenado ou pode ser usado. Neste caso, estamos dando ao nosso campo dois argumentos:

* max\_length=20 — Afima que o valor máximo do comprimento desse campo é de 20 caracteres.
* help\_text='Enter field documentation' — fornece um rótulo de texto para exibir uma ajuda para os usuários saberem qual valor fornecer, quando esse valor é inserido por um usuário por meio de um formulário HTML.

O nome do campo é usado para se referir a ele em consultas e modelos. Os campos também têm um rótulo, que é especificado como um argumento (verbose\_name) ou inferido ao capitalizar a primeira letra do nome da variável do campo e substituindo quaisquer sublinhados por um espaço (por exemplo, my\_field\_name teria um rótulo padrão de My field name).

A ordem em que os campos são declarados afetará sua ordem padrão, se um modelo for representado em um formulário (por exemplo, no site Admin), embora isso possa ser substituído.

##### ARGUMENTOS COMUNS DE UM CAMPO

Os seguintes argumentos são comuns e podem ser usados quando declaramos muitos ou a maioria dos diferentes tipos de campos:

* [help\_text](https://docs.djangoproject.com/en/2.1/ref/models/fields/#help-text): Fornece um rótulo de texto para formulários HTML (por exemplo, no site admin), conforme descrito acima.
* [verbose\_name](https://docs.djangoproject.com/en/2.1/ref/models/fields/#verbose-name): Um nome legível para o campo usado nos rótulos de campo. Se não for especificado, o Django irá inferir o nome detalhado do campo name.
* [default](https://docs.djangoproject.com/en/2.1/ref/models/fields/#default): O valor padrão para o campo. Isso pode ser um valor ou um objeto que pode ser chamado. Cada vez que o objeto for chamado será criado um novo registro.
* [null](https://docs.djangoproject.com/en/2.1/ref/models/fields/#null): Se for True, o Django armazenará valores em branco como NULL no banco de dados, para campos onde isso é apropriado (um CharField irá armazenar uma string vazia). O padrão é False.
* [blank](https://docs.djangoproject.com/en/2.1/ref/models/fields/#blank):Se for True, o campo poderá ficar em branco nos seus formulários. O padrão é False, o que significa que a validação de formulário do Django forçará você a inserir um valor. Isso é frequentemente usado com null = True, porque se você permitir valores em branco, também desejará que o banco de dados possa representá-los adequadamente.
* [choices](https://docs.djangoproject.com/en/2.1/ref/models/fields/#choices): Um grupo de escolhas para este campo. Se isso for fornecido, o padrão widget de formulário correspondente será uma caixa de seleção com essas opções, em vez do campo de texto padrão.
* [primary\_key](https://docs.djangoproject.com/en/2.1/ref/models/fields/#primary-key):Se True, define o campo atual como a chave primária do modelo (uma chave primária é uma coluna especial do banco de dados, designada para identificar exclusivamente as diferentes tabelas) . Se nenhum campo for especificado como a chave primária, o Django adicionará automaticamente um campo para essa finalidade.

Existem muitas outras opções - você pode ver [a lista completa de opções aqui](https://docs.djangoproject.com/en/2.1/ref/models/fields/#field-options).

##### TIPOS COMUNS DE UM CAMPO

A lista a seguir descreve alguns dos tipos de campos mais usados.

* [CharField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.CharField) é usado para definir um tamanho fixo (médio a curto) para a string. Você deve especificar o max\_length (tamanho máximo) para o dado que será armazenado.
* [TextField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.TextField) é usado para grandes strings de comprimento variado. Você pode especificar um max\_length (tamanho máximo) para o campo, mas isso é usado somente quando o campo é exibido em formulários (forms) (ele não é imposto no nível do banco de dados).
* [IntegerField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.IntegerField) é um campo para armazenar números inteiros e para validar valores inseridos como números inteiros em formulários.
* [DateField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#datefield) e [DateTimeField](https://docs.djangoproject.com/en/2.1/ref/models/fields/" \l "datetimefield) são usados para armazenar / representar datas e informações de data / hora (como os objetos Python datetime.date in e datetime.datetime, respectivamente). Esses campos também podem declarar os parâmetros (mutuamente exclusivos) auto\_now = True (para definir o campo para a data atual toda vez que o modelo é salvo), auto\_now\_add (para definir a data em que o primeiro modelo foi criado) e default (para definir uma data padrão que pode ser substituída pelo usuário).
* [EmailField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#emailfield) é usada para armazenara e validar em endereço de email.
* [FileField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#filefield) e [ImageField](https://docs.djangoproject.com/en/2.1/ref/models/fields/" \l "imagefield) são usados para carregar arquivos e imagens respectivamente, (o ImageField simplesmente valida de forma adicional que o arquivo enviado é uma imagem). Eles têm parâmetros para definir como e onde os arquivos enviados são armazenados.
* [AutoField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#autofield) é um tipo especial de IntegerField que é incrementada automaticamente. Uma chave primária desse tipo é adicionada de forma automatica ao seu modelo, se você não especificar explicitamente um.
* [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#foreignkey) é usado para especificar um relacionamento um-para-muitos com outro modelo do banco de dados (por exemplo, um carro tem um fabricante, mas um fabricante pode fazer muitos carros). O lado "um" do relacionamento é o modelo que contém a "chave" (os modelos que contêm uma "chave estrangeira" referem-se a essa "chave" e estão no lado "muitos" de tal relacionamento).
* [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#manytomanyfield) é usado para especificar um relacionamento muitos-para-muitos (por exemplo, um livro pode ter vários gêneros e cada gênero pode conter vários livros). Em nosso aplicativo de biblioteca, usaremos isso de maneira muito semelhante às ForeignKeys, mas elas podem ser usadas de maneiras mais complicadas para descrever as relações entre os grupos. Eles têm o parâmetro on\_delete para definir o que acontece quando o registro associado é excluído (por exemplo, um valor de models.SET\_NULL simplesmente definiria o valor como NULL).

Existem muitos outros tipos de campos, incluindo campos para diferentes tipos de números (big integers, small integers, floats), booleanos, URLs, slugs, unique ids e outras informações "relacionadas ao tempo" (duração, tempo, etc.) . Você pode ver a [lista completa AQUI](https://docs.djangoproject.com/en/2.1/ref/models/fields/#field-types).

#### Metadados (metadada)

Você pode declarar o nível de modelo para os metadados declarando class Meta, como mostrado.

class Meta:

ordering = ['-my\_field\_name']

Copy to Clipboard

Um dos recursos mais úteis desses metadados é controlar a ordem padrão dos registros retornados, quando você consulta o tipo de modelo. Você faz isso especificando a ordem de correspondência em uma lista de nomes para ordenar (ordering) o atributo , conforme mostrado acima. A ordem dependerá do tipo de campo (os campos de caractere são classificados em ordem alfabética, enquanto os campos de data são classificados em ordem cronológica). Como mostrado acima, você pode prefixar o nome do campo com um símbolo de menos (-) para inverter a ordem de classificação.

Então, como exemplo, se optássemos por ordenar livros como este por padrão:

ordering = ['title', '-pubdate']

Copy to Clipboard

Os livros seriam classificados em ordem alfabética por título, de A-Z e depois por data de publicação dentro de cada título, do mais recente ao mais antigo.

Outro atributo comum é verbose\_name, um nome detalhado para a classe no singular e plural:

verbose\_name = 'BetterName'

Copy to Clipboard

Outros atributos úteis permitem que você crie e aplique novas "permissões de acesso" para o modelo (as permissões padrão são aplicadas automaticamente), permitem a ordenação com base em outro campo ou declarar que a classe é "abstrata" (uma classe base que você não pode criar registros, e em vez disso, serão derivadas para criar outros modelos).

Muitas das outras opções de metadados controlam qual banco de dados deve ser usado para o modelo e como os dados são armazenados (eles são realmente úteis somente se você precisar mapear um modelo para um banco de dados existente).

A lista completa de opções de metadados pode ser encontrada aqui: [Opções de modelos de metadados](https://docs.djangoproject.com/en/2.1/ref/models/options/) (Django docs).

#### Métodos

Um modelo também pode ter métodos.

**Minimamente, em cada modelo você deve definir o método de classe padrão do Python \_\_str\_\_() para retornar uma string legível para cada objeto.** Essa sequência é usada para representar registros individuais no site de administração (e em qualquer outro lugar que você precise se referir a uma instância de modelo). Muitas vezes isso retornará um campo de título ou nome do modelo.

def \_\_str\_\_(self):

return self.field\_name

Copy to Clipboard

Outro método comum a incluir nos modelos do Django é o get\_absolute\_url(), que retorna uma URL para exibir registros de modelos individuais no site (se você definir esse método, o Django adicionará automaticamente um botão "View on Site" às ​​telas de edição de registros do modelo o site Admin). Um padrão típico para get\_absolute\_url () é mostrado abaixo.

def get\_absolute\_url(self):

"""Retorna o URL para acessar uma instância específica do modelo."""

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

Copy to Clipboard

**Nota:**Supondo que você usará URLs como / myapplication / mymodelname / 2 para exibir registros individuais para seu modelo (onde "2" é o id para um registro específico), você precisará criar um mapeador de URL para passar a resposta e id para uma "vista detalhada do modelo" (que fará o trabalho necessário para exibir o registro). A função reverse () acima é capaz de "inverter" seu mapeador de url (no caso acima chamado 'model-detail-view') para criar uma URL do formato correto.

Claro que para fazer este trabalho você ainda tem que escrever o mapeamento de URL, visão e modelo!

Você também pode definir outros métodos que desejar e chamá-los de seu código ou modelos (desde que eles não utilizem nenhum parâmetro).

### [Gestão de modelos](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#gest%C3%A3o_de_modelos)

Depois de definir suas classes de modelo, você pode usá-las para criar, atualizar ou excluir registros e executar consultas para obter todos os registros ou subconjuntos específicos de registros. Mostraremos como fazer isso no tutorial quando definirmos nossas visualizações, mas aqui está um breve resumo.

#### Criando e modificando registros

Para criar um registro, você pode definir uma instância do modelo e, em seguida, chamar save ().

# Crie um novo registro usando o construtor do modelo.

record = MyModelName(my\_field\_name="Instance #1")

# Salve o objeto no banco de dados.

record.save()

Copy to Clipboard

**Nota:** Se você não tiver declarado qualquer campo como primary\_key, o novo registro receberá um automaticamente, com o ID do nome do campo. Você poderia consultar este campo depois de salvar o registro acima e ele teria um valor de 1.

Você pode acessar os campos nesse novo registro usando a sintaxe de ponto e alterar os valores. Você precisa chamar save () para armazenar valores modificados no banco de dados.

# Acessar valores de campo de modelo usando atributos do Python.

print(record.id) # should return 1 for the first record.

print(record.my\_field\_name) # should print 'Instance #1'

# Altere o registro modificando os campos e, em seguida, chamando save ().

record.my\_field\_name = "New Instance Name"

record.save()

Copy to Clipboard

#### Procurando por registros

Você pode procurar registros que correspondam a um determinado critério usando o atributo de objetos do modelo (fornecido pela classe base).

**Nota:**Explicar como procurar registros usando nomes de campos e modelos "abstratos" pode ser um pouco confuso. Na discussão abaixo, vamos nos referir a um modelo de livro com campos de título e gênero, onde o gênero também é um modelo com um único nome de campo.

Podemos obter todos os registros para um modelo como um QuerySet, usando objects.all (). O QuerySet é um objeto iterável, o que significa que ele contém um número de objetos que podemos iterar / percorrer.

all\_books = Book.objects.all()

Copy to Clipboard

O método filter () do Django nos permite filtrar o QuerySet retornado para corresponder a um campo especificado de **texto** ou **numérico** em relação a um critério específico. Por exemplo, para filtrar por livros que contenham "wild" no título e, em seguida, contá-los, poderíamos fazer o seguinte.

wild\_books = Book.objects.filter(title\_\_contains='wild')

number\_wild\_books = Book.objects.filter(title\_\_contains='wild').count()

Copy to Clipboard

Os campos a serem correspondidos e o tipo de correspondência são definidos no nome do parâmetro de filtro, usando o formato: field\_name\_\_match\_type (observe o sublinhado duplo entre título e contém acima). Acima, estamos filtrando o título com uma correspondência de maiúsculas e minúsculas. Existem muitos outros tipos de correspondência que você pode fazer: icontains (insensitivo a maiúsculas e minúsculas), iexact (correspondência exata que não diferencia maiúsculas e minúsculas), exata (correspondência exata diferencia maiúsculas de minúsculas), gt (maior que), startswith, etc. é [aqui](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#field-lookups).

Em alguns casos, você precisará filtrar um campo que defina um relacionamento um-para-muitos com outro modelo (por exemplo, uma ForeignKey). Nesse caso, você pode "indexar" campos no modelo relacionado com sublinhados duplos adicionais. Por exemplo, para filtrar livros com um padrão de gênero específico, você terá que indexar o nome por meio do campo de gênero, conforme mostrado abaixo:

# Combinará em: ficção, ficção científica, não-ficção etc.

books\_containing\_genre = Book.objects.filter(genre\_\_name\_\_icontains='fiction')

Copy to Clipboard

**Nota:** Você pode usar sublinhados (\_\_) para navegar quantos níveis de relacionamentos (ForeignKey / ManyToManyField) desejar. Por exemplo, um Livro que tinha tipos diferentes, definidos usando um relacionamento "cover" adicional, pode ter um nome de parâmetro: type\_\_cover\_\_name\_\_exact = 'hard'.

Há muito mais que você pode fazer com as consultas, incluindo pesquisas para trás de modelos relacionados, encadeando filtros, retornando um conjunto menor de valores, etc. Para obter mais informações, consulte [Fazendo consultas](https://docs.djangoproject.com/en/2.1/topics/db/queries/) (Django Docs).

## [Definindo os modelos LocalLibrary](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#definindo_os_modelos_locallibrary)

Nesta seção, começaremos a definir os modelos para a biblioteca. Abra models.py (em / locallibrary / catalog /). O código na parte superior da página importa o módulo de models, que contém os modelos da classe base do models. Modelo do qual nossos models herdarão.

from django.db import models

# Create your models here.

Copy to Clipboard

### [Genre model](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#genre_model)

Copie o código do model Genre , que aparece a baixo, e cole no seu arquivo models.py , abaixo do código de importação.  Esse model  guarda informaçoes sobre a categoria do livro — por exemplo se é ficção ou não ficção, romance ou história, etc. Como mencionado acima, criamos o genero como um model em vez de um campo de texto ou um lista de seleção para que os possíveis generos criados possam ser gerenciados pelo banco de dados, em vez de serem codificados.

class Genre(models.Model):

"""Model representing a book genre."""

name = models.CharField(max\_length=200, help\_text='Enter a book genre (e.g. Science Fiction)')

def \_\_str\_\_(self):

"""String for representing the Model object."""

return self.name

O model tem apenas um campo do tipo CharField (name), que é usado para descrever o genero (esse campo é limitado a 200 caracteres e tem um help\_text). No final da classe Genre declaramos o metodo  \_\_str\_\_() que retorna o nome do genero definido por um registro especifico. Não foi definido um argumento verbose\_name , assim o campo será referenciado como  Name nos forms.

### [Book model](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#book_model)

Copie o model Book abaixo e cole novamente ao final do seu arquivo. O model Book representa todas as informações disponíveis sobre um livro de maneira geral, mas não incluí detalhes sobre o formato do encadernamento ou disponibilidade para empréstimo. O model usa um CharField para representar o  title (tílutlo) do livro e o isbn (veja que o isbn recebe um rótulo como "ISBN", usando o primeiro parâmetro não nomeado, porque, de outra forma, o rótulo ficaria "Isbn"). O model usa TextField para o campo summary porque ele precisa ser um tanto longo.

from django.urls import reverse # Used to generate URLs by reversing the URL patterns

class Book(models.Model):

    """Model representing a book (but not a specific copy of a book)."""

    title = models.CharField(max\_length=200)

# Foreign Key used because book can only have one author, but authors can have multiple books

# Author as a string rather than object because it hasn't been declared yet in the file

    author = models.ForeignKey('Author', on\_delete=models.SET\_NULL, null=True)

    summary = models.TextField(max\_length=1000, help\_text='Enter a brief description of the book')

    isbn = models.CharField('ISBN', max\_length=13, help\_text='13 Character <a href="https://www.isbn-international.org/content/what-isbn">ISBN number</a>')

    # ManyToManyField used because genre can contain many books. Books can cover many genres.

    # Genre class has already been defined so we can specify the object above.

genre = models.ManyToManyField(Genre, help\_text='Select a genre for this book')

    def \_\_str\_\_(self):

        """String for representing the Model object."""

        return self.title

    def get\_absolute\_url(self):

        """Returns the url to access a detail record for this book."""

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

Genre é um campo ManyToManyField, de forma que um livro pode ter múltiplos gêneros e um gênero pode ter muitos livros. O campor Author é declarado como um ForeignKey, logo, cada livro terá somente um autor, mas um autor por der muitos livros (na prática, um livro pode ter múltiplos autores, mas não nesta implementação!)

Em ambos tipos de campo, a classe model relacionada é declarada como primeiro campo não nomeado usando a classe model ou contendo o nome do modelo relacionado. Você deve usar o nome do model como uma string se a classe associada ainda não tiver sido definida no arquivo, antes de referenciá-la. Outro parâmetro de interesse no campo author é null=True, o qual permite ao banco de dados armazer um valor Null se nenhum autor for selecionado, e, on\_delete=models.SET\_NULL, o qual definirá o valor author como Null se o registro do autor associado for excluído.

O mode também defini \_\_str\_\_(), usando o title do livro para representar o registro do Book. O método final, get\_absolute\_url(), retorna a URL que pode ser usada para acessar o registro detalhado deste modelo (para que isto funcione, teremos que definir um mapa de URL com o nome book-detail e associá-la a uma view e a um template.).

### [BookInstance model](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#bookinstance_model)

Em seguida, copie o modelo BookInstance (exibido a abaixo) depois dos outros modelos. O BookInstance representa um exemplar específico do livro que pode ser emprestado, indica se ela está disponível ou a data programada de restituição, "imprint" ou detalhes da versão, e, um id único do livro na biblioteca.

Alguns dos campos e métodos serão familiares agora. O model usa

* ForeignKey para identificar o Book associado (cada livro pode ter muitos exemplares, porém uma cópia pode ter sometne um Book).
* CharField para representar o imprint (edição específica) do livro.

import uuid # Required for unique book instances

class BookInstance(models.Model):

"""Model representing a specific copy of a book (i.e. that can be borrowed from the library)."""

id = models.UUIDField(primary\_key=True, default=uuid.uuid4, help\_text='Unique ID for this particular book across whole library')

book = models.ForeignKey('Book', on\_delete=models.SET\_NULL, null=True)

imprint = models.CharField(max\_length=200)

due\_back = models.DateField(null=True, blank=True)

LOAN\_STATUS = (

('m', 'Maintenance'),

('o', 'On loan'),

('a', 'Available'),

('r', 'Reserved'),

)

status = models.CharField(

max\_length=1,

choices=LOAN\_STATUS,

blank=True,

default='m',

help\_text='Book availability',

)

class Meta:

ordering = ['due\_back']

def \_\_str\_\_(self):

"""String for representing the Model object."""

return f'{self.id} ({self.book.title})'

Copy to Clipboard

Nós declaramos adicionalmente alguns tipos novos de campos:

* UUIDField é usado no campo id para definí-lo como primary\_key deste model. Este tipo de campo aloca um valor único global para cada instância (um para cada livro encontrado na biblioteca).
* DateField é usado para a data due\_back (na qual o livro deve retonar e ficar disponível após um empréstimo ou uma manutenção). O valor pode ser blank ou null (para quando o livro estiver disponível). O model metadata (Class Meta) usa este campo para ordenar os registros quando eles são retornados em uma query.
* status é um CharField que define uma lista de opção/seleção. Como pode ser visto, nós definimos uma dupla contendo pares de valores-chave e a passamos no argumento da opção. O valor do par valor-chave é um valor que o usuário pode selecionar, enquanto que a chave é o valor que  realmetne será salvo se a opção for selecionada. Nós também definimos um valor default como "m" (maintenance), uma vez que os livros serão inicialmente criados como indisponíveis até que eles sejam disponibilizados nas prateleiras.

O model \_\_str\_\_() representa o objeto BookInstance usando a combinação do seu id único e o título do Book associado.

**Nota**: Um pouco sobre Python:

* Começando com Python 3.6, pode ser usada a sintaxe de interpolação  (também conhecida como f-strings): f'{self.id} ({self.book.title})'.
* Em versões anteriores deste tutorial, nós estávamos usando a sintaxe de  [formatted string](https://www.python.org/dev/peps/pep-3101/), a qual também é uma maneira válida de formatar  strings em Python (e.g. '{0} ({1})'.format(self.id,self.book.title)).

### [Author model](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#author_model)

Copie o model Author (exibido abaixo) na sequência do código existente em  **models.py**.

Todos os campos/métodos já devem ser familiares agora. O model define um autor como tendo um nome, sobrenome, data de nascimento e data de morte (opcionalmente). Ele especifica que \_\_str\_\_() retorna por default sobrenome e nome, nesta ordem. O método get\_absolute\_url() reverte o mapaeamento da URL author-detail para obter a URL de display individual author.

class Author(models.Model):

"""Model representing an author."""

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)

class Meta:

ordering = ['last\_name', 'first\_name']

def get\_absolute\_url(self):

"""Returns the url to access a particular author instance."""

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

def \_\_str\_\_(self):

"""String for representing the Model object."""

return f'{self.last\_name}, {self.first\_name}'

Copy to Clipboard

## [Rode novamente a migração do banco de dados](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#rode_novamente_a_migra%C3%A7%C3%A3o_do_banco_de_dados)

Todos os modelos foram criados. Agora rode a migração do banco de dados para adicioná-los ao seu banco de dados.

python3 manage.py makemigrations

python3 manage.py migrate

## [Language model — desafio](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models#language_model_%E2%80%94_desafio)

Considere que um benfeitor local doe uma quantidade de livros escritos em outro idioma (digamos, Farsi). O desafio é desenvolver como eles seriam melhor representados no website da nossa biblioteca, e, então, adicioná-lo no models.

Algumas coisas a considerar:

* O idioma seria associado ao  Book, BookInstance, ou algum outro objeto?
* Os diferentes idiomas seriam representados usando model, como campo de texto livre ou codificado como uma lista de seleção?

Depois de decidido, adicione o campo. Pode ser visto o que decidimos no Github [aqui](https://github.com/mdn/django-locallibrary-tutorial/blob/master/catalog/models.py).

## [Registrando modelos](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#registrando_modelos)

Primeiro, abra o admin.py no aplicativo de catálogo (/locallibrary/catalog/admin.py). Atualmente parece com isso - note que ele já importa django.contrib.admin:

from django.contrib import admin

# Register your models here.

Copy to Clipboard

Registre os modelos copiando o seguinte texto na parte inferior do arquivo. Este código simplesmente importa os modelos e, em seguida, chama admin.site.register para registrar cada um deles.

from catalog.models import Author, Genre, Book, BookInstance

admin.site.register(Book)

admin.site.register(Author)

admin.site.register(Genre)

admin.site.register(BookInstance)

Copy to Clipboard

Nota: Se você aceitou o desafio de criar um modelo para representar a linguagem natural de um livro ([consulte o artigo do tutorial de modelos](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Models)), importe-o e registre-o também!

Essa é a maneira mais simples de registrar um modelo ou modelos no site. O site de administração é altamente personalizável e falaremos mais sobre as outras maneiras de registrar seus modelos mais abaixo.

## [Criando um super usuário](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#criando_um_super_usu%C3%A1rio)

Para fazer login no site de administração, precisamos de uma conta de usuário com o status da equipe ativado. Para visualizar e criar registros, também precisamos que esse usuário tenha permissões para gerenciar todos os nossos objetos. Você pode criar uma conta "superusuário" que tenha acesso total ao site e todas as permissões necessárias usando **manage.py**.

Chame o seguinte comando, no mesmo diretório que manage.py, para criar o superusuário. Você será solicitado a digitar um nome de usuário, endereço de e-mail e senha forte.

python3 manage.py createsuperuser

Copy to Clipboard

Quando esse comando for concluído, um novo superusuário será adicionado ao banco de dados. Agora reinicie o servidor de desenvolvimento para que possamos testar o login:

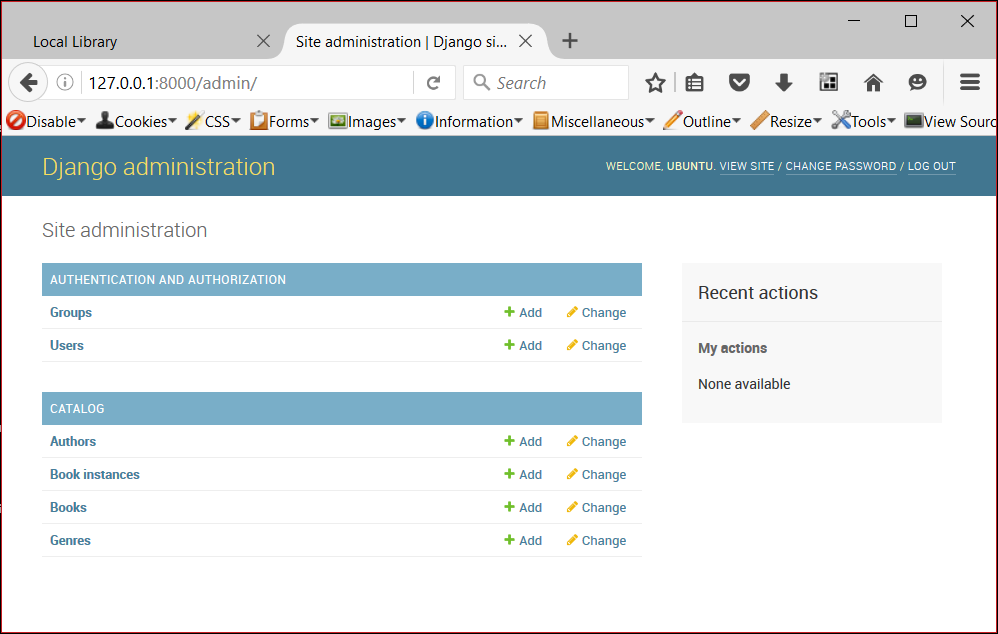
python3 manage.py runserver

Copy to Clipboard

## [Fazendo o login e usando o site](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#fazendo_o_login_e_usando_o_site)

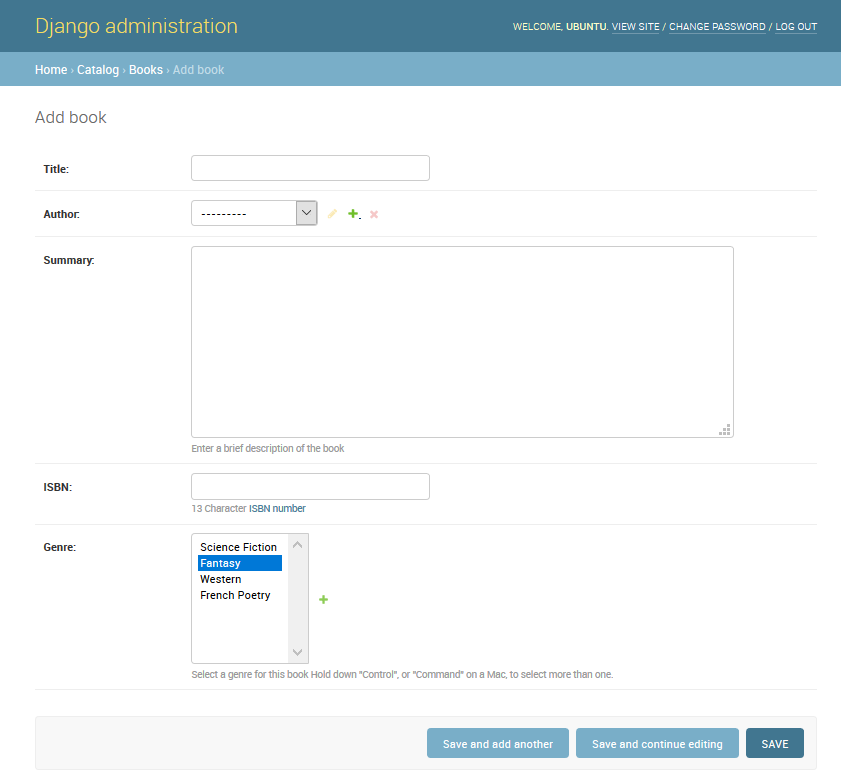
Para fazer login no site, abra o URL /admin (e.g. [http://127.0.0.1:8000/admin](http://127.0.0.1:8000/admin/)) e insira suas novas credenciais de usuário e senha de superusuário (você será redirecionado para a página de login e, em seguida, de volta para o URL /admin depois de inserir seus detalhes).

Esta parte do site exibe todos os nossos modelos, agrupados por aplicativo instalado. Você pode clicar no nome de um modelo para ir a uma tela que lista todos os seus registros associados e clicar nos registros para editá-los. Você também pode clicar diretamente no link Adicionar ao lado de cada modelo para começar a criar um registro desse tipo.



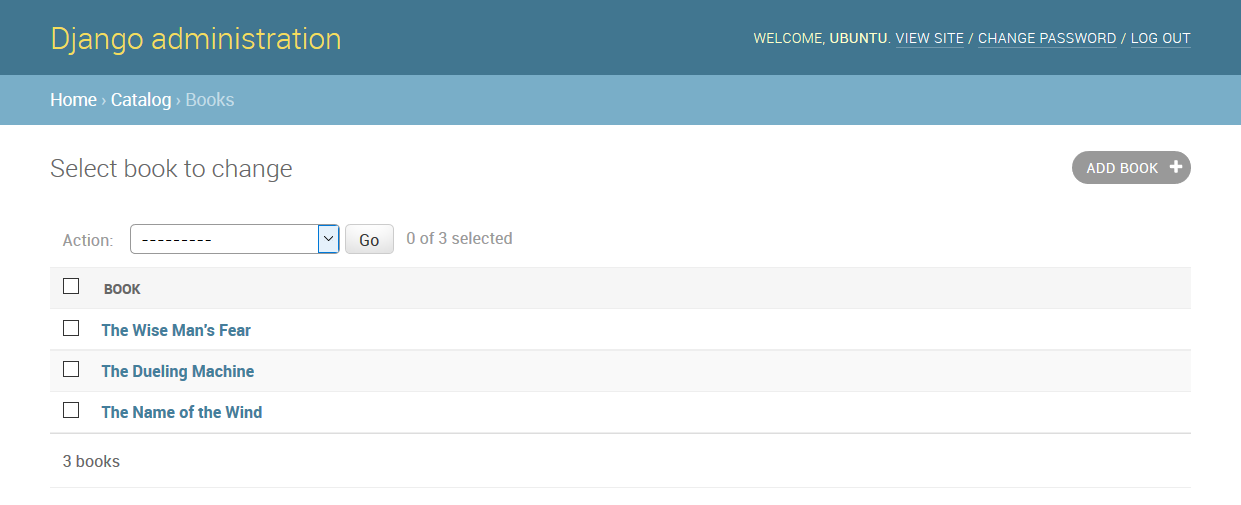
Clique no link Adicionar à direita de Books para criar um novo livro (isso exibirá um diálogo muito parecido com o abaixo). Observe como os títulos de cada campo, o tipo de widget usado e o help\_text (se houver) correspondem aos valores especificados no modelo.

Digite valores para os campos. Você pode criar novos autores ou gêneros pressionando o botão + ao lado dos respectivos campos (ou selecione os valores existentes nas listas, se você já os criou). Quando estiver pronto, você pode pressionar**SALVAR, Salvar** **e adicionar**outro ou S**alvar e continuar editando** para salvar o registro.



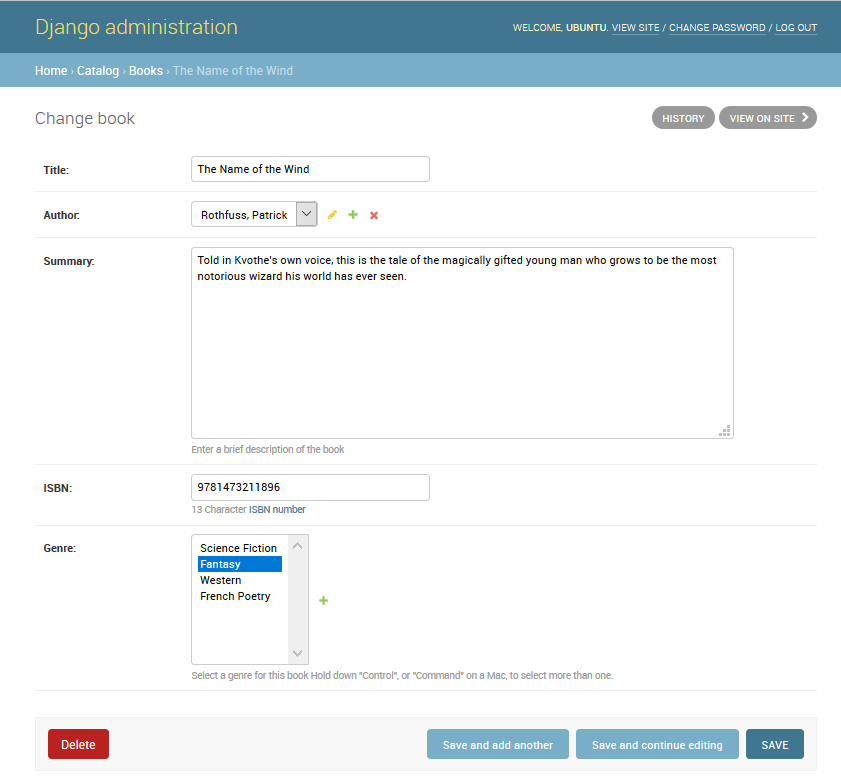
Observação: neste ponto, gostaríamos que você passasse algum tempo adicionando alguns livros, autores e gêneros (por exemplo, Fantasia) à sua inscrição. Certifique-se de que cada autor e gênero inclua alguns livros diferentes (isso tornará suas visualizações de lista e detalhes mais interessantes quando forem implementadas posteriormente na série de artigos).

Quando terminar de adicionar livros, clique no link **Home** no marcador superior para ser levado de volta à página principal do administrador. Então clique no link **Books**para exibir a lista atual de livros (ou em um dos outros links para ver outras listas de modelos). Agora que você adicionou alguns livros, a lista pode ser semelhante à captura de tela abaixo. O título de cada livro é exibido; este é o valor retornado no modelo do livro pelo método \_\_str\_\_() que especificamos no último artigo.



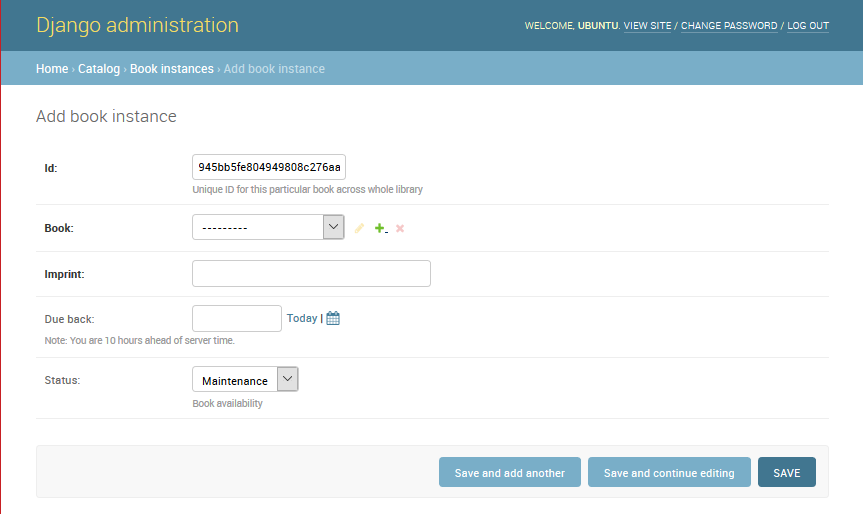
Nessa lista, você pode excluir livros marcando a caixa de seleção ao lado do livro que não deseja, selecionando a ação excluir ... na lista suspensa Ação e pressionando o botão Ir. Você também pode adicionar novos livros pressionando o botão **ADD BOOK**.

Você pode editar um livro selecionando seu nome no link. A página de edição de um livro, mostrada abaixo, é quase idêntica à página "Adicionar". As principais diferenças são o título da página (Change book) e a adição de botões **Delete**, **HISTORY** e **VIEW ON SITE** (este último botão aparece porque definimos o método get\_absolute\_url() em nosso modelo).



Agora navegue de volta para o **Home** page (usando o link Home, a trilha de navegação) e, em seguida, **Author** e listas de **Genre** — você já deve ter criado a partir de quando adicionou os novos livros, mas fique à vontade para adicionar um pouco mais.

O que você não terá é qualquer instância do livro, porque elas não são criadas a partir de livros (embora você possa criar Book a partir de BookInstance — esta é a natureza da ForeignKey field). Navegue de volta para a Página inicial e pressione o botão Adicionar associado para exibir a tela Adicionar instância do livro abaixo. Observe o ID grande e globalmente exclusivo, que pode ser usado para identificar separadamente uma única cópia de um livro na biblioteca.



Crie vários desses registros para cada um de seus livros. Defina o status como Disponível para pelo menos alguns registros e Em empréstimo para outros. Se o status **não**for Disponível, defina também uma data de vencimento futura.

É isso aí! Agora você aprendeu como configurar e usar o site de administração. Você também criou registros para Book, BookInstance, Genre, e Author que poderemos usar assim que criarmos nossas próprias visualizações e modelos.

## [Configuração Avançada](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#configura%C3%A7%C3%A3o_avan%C3%A7ada)

O Django faz um bom trabalho ao criar um site de administração básico usando as informações dos modelos registrados:

* Cada modelo possui uma lista de registros individuais, identificados pela string criada com o método \_\_str\_\_() do modelo, e vinculado a visualizações de views/forms para edição. Por padrão, essa exibição tem um menu de ação na parte superior que você pode usar para executar operações de exclusão em massa nos registros.
* Os formulários de registro de detalhes do modelo para edição e adição de registros contêm todos os campos do modelo, dispostos verticalmente em sua ordem de declaração.

Você pode personalizar ainda mais a interface para torná-la ainda mais fácil de usar. Algumas das coisas que você pode fazer são:

* List views:
  + Adicionar adicional fields/information exibido para cada registro.
  + Adicione filtros para selecionar quais registros são listados, com base na data ou em algum outro valor de seleção (e.g. Book loan status).
  + Adicione opções adicionais ao menu de ações nas exibições de lista e escolha onde esse menu é exibido no formulário.
* Detail views
  + Escolha quais campos exibir (ou excluir), junto com sua ordem, agrupamento, se eles são editáveis, o widget usado, a orientação etc.
  + Adicione campos relacionados a um registro para permitir a edição imediata (por exemplo, adicione a capacidade de adicionar e editar registros de livros enquanto cria o registro de autor).

In this section we're going to look at a few changes that will improve the interface for our LocalLibrary, including adding more information to Book and Author model lists, and improving the layout of their edit views. We won't change the Language and Genre model presentation because they only have one field each, so there is no real benefit in doing so!

You can find a complete reference of all the admin site customisation choices in [The Django Admin site](https://docs.djangoproject.com/en/2.0/ref/contrib/admin/) (Django Docs).

### [Registrando uma classe ModelAdmin](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#registrando_uma_classe_modeladmin)

Para alterar como um modelo é exibido na interface de administração, você define uma classe ModelAdmin (que descreve o layout) e registra-o no modelo.

Vamos começar com o Author model. Abra **admin.py** no aplicativo de catálogo (/locallibrary/catalog/admin.py). Comente o seu registro original (prefixo com um #) para o Author model:

# admin.site.register(Author)

Copy to Clipboard

Agora adicione um novo AuthorAdmin e registre como mostrado abaixo.

# Define the admin class

class AuthorAdmin(admin.ModelAdmin):

pass

# Register the admin class with the associated model

admin.site.register(Author, AuthorAdmin)

Copy to Clipboard

Agora vamos adicionar as classes ModelAdmin para Book, e BookInstance. Precisamos novamente comentar os registros originais:

# admin.site.register(Book)

# admin.site.register(BookInstance)

Copy to Clipboard

Agora, para criar e registrar os novos modelos; para o propósito desta demonstração, vamos usar o @register decorador para registrar os modelos (isso faz exatamente a mesma coisa que admin.site.register() sintaxe):

# Register the Admin classes for Book using the decorator

@admin.register(Book)

class BookAdmin(admin.ModelAdmin):

pass

# Register the Admin classes for BookInstance using the decorator

@admin.register(BookInstance)

class BookInstanceAdmin(admin.ModelAdmin):

pass

Copy to Clipboard

Atualmente todas as nossas classes administrativas estão vazias (veja pass) então o comportamento do administrador não será alterado! Agora podemos estendê-los para definir nosso comportamento administrativo específico do modelo.

### [Configure list views](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#configure_list_views)

A LocalLibrary atualmente lista todos os autores usando o nome do objeto gerado a partir do método \_\_str\_\_() do modelo. Isso é bom quando você tem apenas alguns autores, mas quando você tem muitos, você pode acabar tendo duplicatas. Para diferenciá-los, ou apenas porque você quer mostrar informações mais interessantes sobre cada autor, você pode usar [list\_display](https://docs.djangoproject.com/en/dev/ref/contrib/admin/" \l "django.contrib.admin.ModelAdmin.list_display) para adicionar campos adicionais à vista.

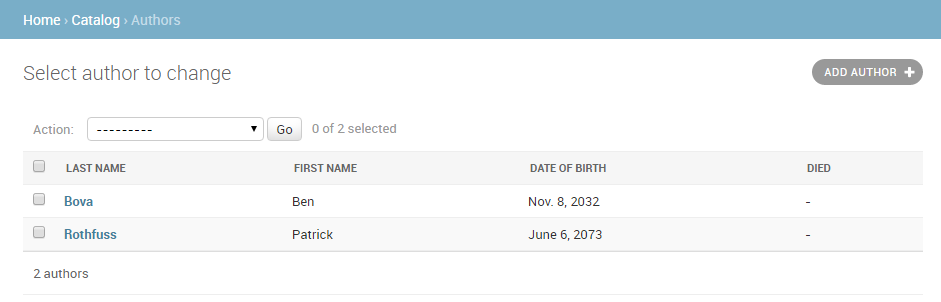
Substitua seuAuthorAdmin class com o código abaixo. Os nomes de campo a serem exibidos na lista são declarados em uma tupla na ordem requerida, conforme mostrado (esses são os mesmos nomes especificados em seu modelo original).

class AuthorAdmin(admin.ModelAdmin):

list\_display = ('last\_name', 'first\_name', 'date\_of\_birth', 'date\_of\_death')

Copy to Clipboard

Agora navegue até a lista de autores em seu site. Os campos acima devem agora ser exibidos, assim:



Para o nosso Book model nós vamos adicionalmente exibir o author e genre. O author é uma variável ForeignKey (um-para-um) relacionamento, e assim será representado pelo valor \_\_str\_\_() para o registro associado. Substitua o BookAdmin class com a versão abaixo.

class BookAdmin(admin.ModelAdmin):

list\_display = ('title', 'author', 'display\_genre')

Copy to Clipboard

Infelizmente não podemos especificar diretamente a variável  genre na list\_display porque é um ManyToManyField(O Django evita isso porque há um grande "custo" de acesso ao banco de dados ao fazer isso). Em vez disso, vamos definir uma função display\_genre para obter as informações como uma string (esta é a função que chamamos acima; vamos defini-lo abaixo).

Nota: Obtendo o genre pode não ser uma boa ideia aqui, por causa do "custo" da operação do banco de dados. Estamos mostrando como as funções de chamada em seus modelos podem ser muito úteis por outros motivos - por exemplo, para adicionar um link Apagar ao lado de cada item da lista.

Adicione o seguinte código ao seu Book model (**models.py**). Isso cria uma string a partir dos três primeiros valores da variavel genre (se existirem) e cria um short\_description que pode ser usado no site administrativo para esse método.

def display\_genre(self):

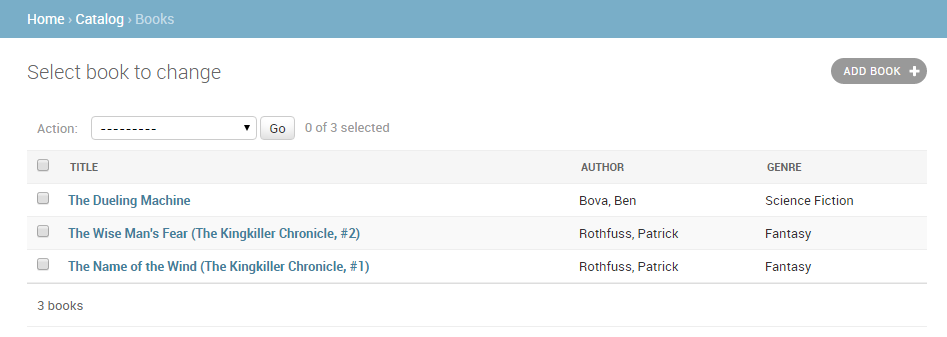
"""Create a string for the Genre. This is required to display genre in Admin."""

return ', '.join(genre.name for genre in self.genre.all()[:3])

display\_genre.short\_description = 'Genre'

Copy to Clipboard

Depois de salvar o modelo e o administrador atualizado, abra o site e vá para a página da lista Livros; você deve ver uma lista de livros como a abaixo:



O Genre model (e a Language model, se você definiu um) ambos têm um único campo, portanto, não faz sentido criar um modelo adicional para exibir campos adicionais.

Nota: Vale a pena atualizar oBookInstance model list para mostrar pelo menos o status e a data de retorno esperada. Nós adicionamos isso como um desafio no final deste artigo!

### [Adicionando list filters](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#adicionando_list_filters)

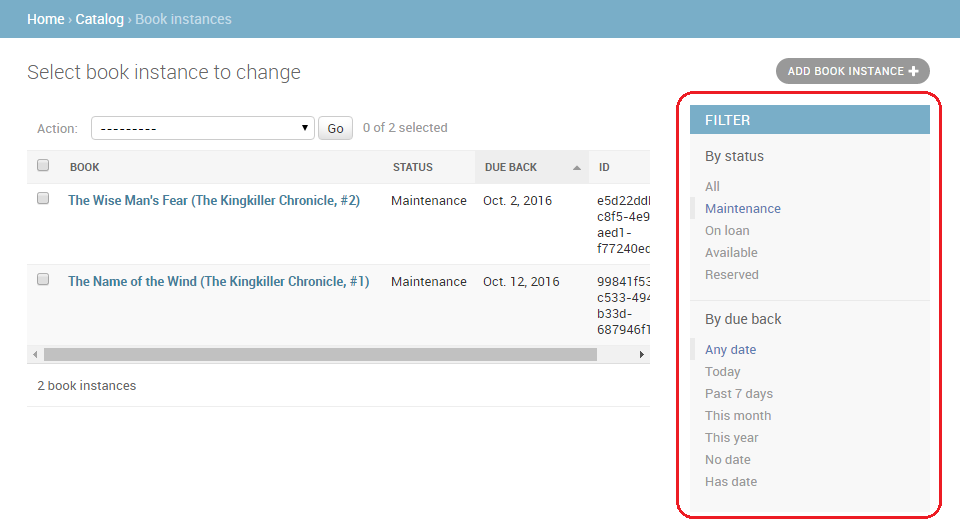
Uma vez que você tenha muitos itens em uma lista, pode ser útil filtrar quais itens são exibidos. Isso é feito listando os campos no atributo list\_filter. Substitua sua atual BookInstanceAdmin class com o fragmento de código abaixo.

class BookInstanceAdmin(admin.ModelAdmin):

list\_filter = ('status', 'due\_back')

Copy to Clipboard

A visualização de lista agora incluirá uma caixa de filtro à direita. Observe como você pode escolher datas e status para filtrar os valores:



### [Organizando o layout da detail view](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#organizando_o_layout_da_detail_view)

Por padrão, as exibições detalhadas exibem todos os campos verticalmente, em sua ordem de declaração no modelo. Você pode alterar a ordem da declaração, quais campos são exibidos (ou excluídos), se as seções são usadas para organizar as informações, se os campos são exibidos horizontalmente ou verticalmente e até mesmo quais widgets de edição são usados nos formulários admin.

Nota: Os modelos LocalLibrary são relativamente simples, portanto não é necessário alterar o layout; No entanto, faremos algumas alterações, só para mostrar como.

#### Controlando quais campos são exibidos

Atualize seu AuthorAdmin class para adicionar a linha fields, como mostrado abaixo (em negrito):

class AuthorAdmin(admin.ModelAdmin):

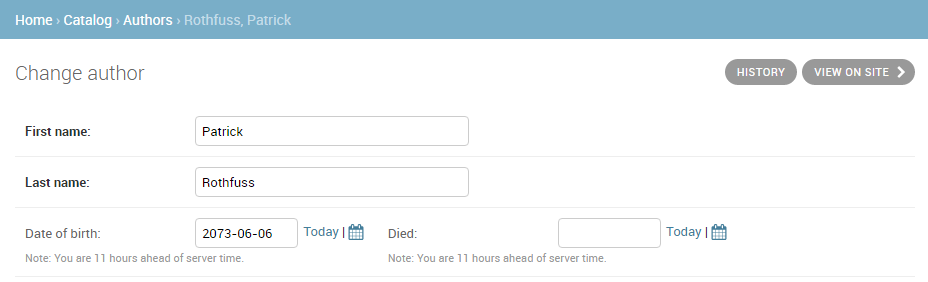
list\_display = ('last\_name', 'first\_name', 'date\_of\_birth', 'date\_of\_death')

fields = ['first\_name', 'last\_name', ('date\_of\_birth', 'date\_of\_death')]

Copy to Clipboard

O atributo fields lista apenas os campos que devem ser exibidos no formulário, em ordem. Os campos são exibidos verticalmente por padrão, mas serão exibidos horizontalmente se você agrupá-los posteriormente em uma tupla (conforme mostrado nos campos "data" acima).

No seu site, acesse a visualização de detalhes do autor. Agora, ele deve aparecer como mostrado abaixo:



Nota: você também pode usar o atributo exclude para declarar uma lista de atributos a serem excluídos do formulário (todos os outros atributos no modelo serão exibidos).

#### Seccionando a detail view

Você pode adicionar "seções" para agrupar informações de modelo relacionadas dentro do formulário detalhado, usando o atributo [fieldsets](https://docs.djangoproject.com/en/dev/ref/contrib/admin/" \l "django.contrib.admin.ModelAdmin.fieldsets).

Na BookInstance model temos informações relacionadas ao que o livro é (i.e. name, imprint, e id) e quando estará disponível (status, due\_back). Podemos adicionar estes em diferentes seções, adicionando o texto em negrito para o nosso BookInstanceAdmin class.

@admin.register(BookInstance)

class BookInstanceAdmin(admin.ModelAdmin):

list\_filter = ('status', 'due\_back')

fieldsets = (

(None, {

'fields': ('book', 'imprint', 'id')

}),

('Availability', {

'fields': ('status', 'due\_back')

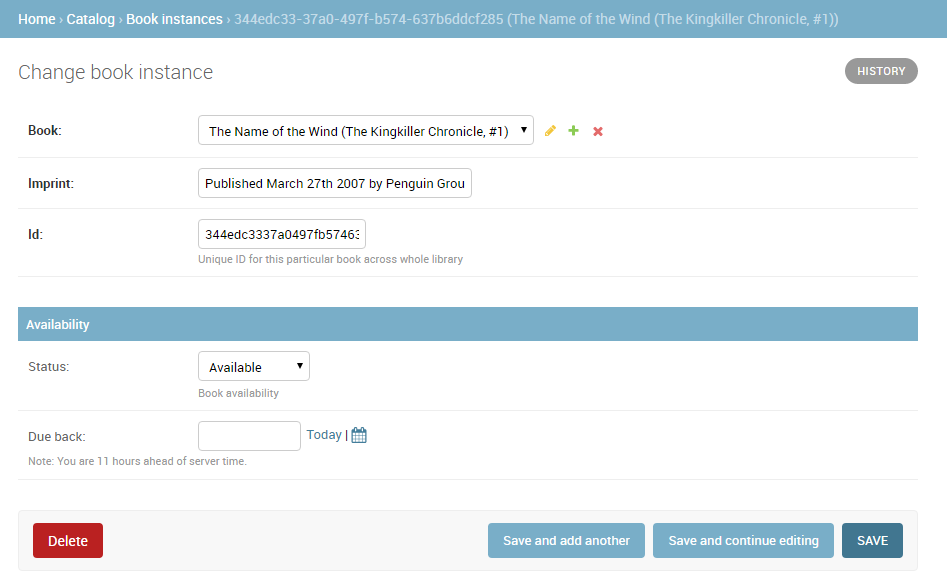
}),

)

Copy to Clipboard

Cada seção tem seu próprio título (ou None,se você não quiser um título) e uma tupla associada de campos em um dicionário - o formato é complicado de descrever, mas bastante fácil de entender se você olhar o fragmento de código imediatamente acima.

Agora, navegue até uma visualização de instância do livro em seu website; o formulário deve aparecer como mostrado abaixo:



### [Edição inline de registros associados](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#edi%C3%A7%C3%A3o_inline_de_registros_associados)

Às vezes, pode fazer sentido adicionar registros associados ao mesmo tempo. Por exemplo, pode fazer sentido ter as informações do livro e as informações sobre as cópias específicas que você tem na mesma página de detalhes.

Você pode fazer isso declarando [inlines](https://docs.djangoproject.com/en/dev/ref/contrib/admin/" \l "django.contrib.admin.ModelAdmin.inlines), do tipo [TabularInline](https://docs.djangoproject.com/en/dev/ref/contrib/admin/" \l "django.contrib.admin.TabularInline) (horizonal layout) or [StackedInline](https://docs.djangoproject.com/en/dev/ref/contrib/admin/" \l "django.contrib.admin.StackedInline) (layout vertical, assim como o layout do modelo padrão). Você pode adicionar ao BookInstance informações inline para o nosso Book detalhe, adicionando as linhas abaixo em negrito perto do seu BookAdmin:

class BooksInstanceInline(admin.TabularInline):

model = BookInstance

@admin.register(Book)

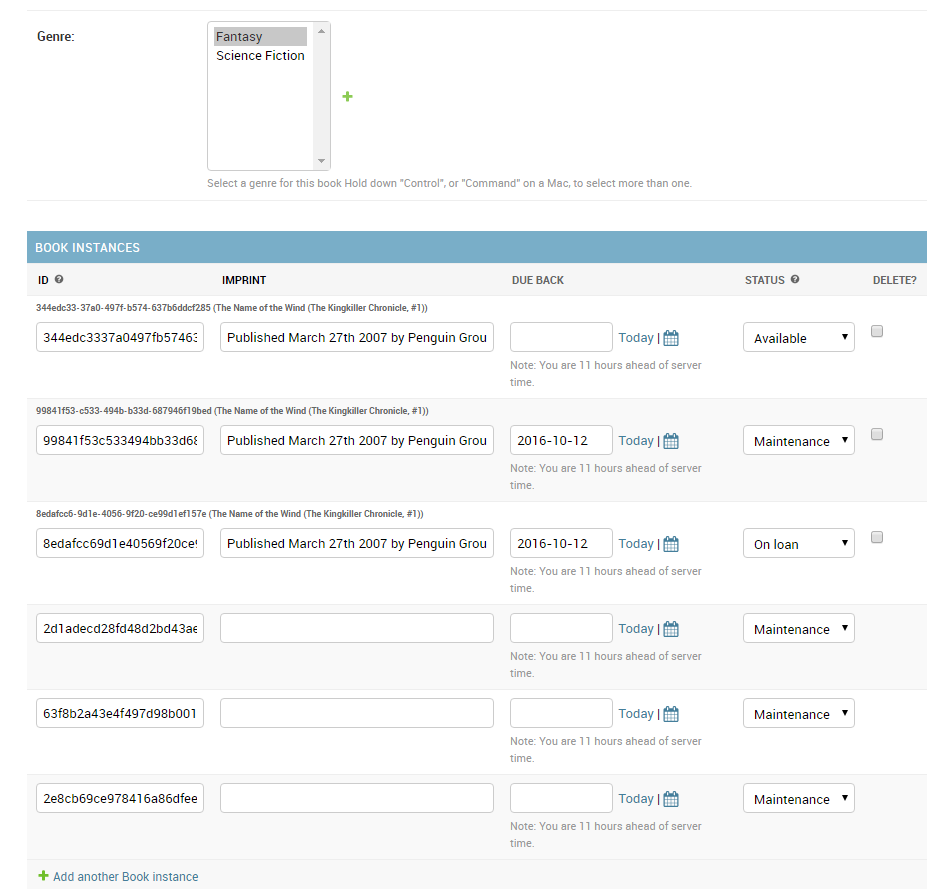
class BookAdmin(admin.ModelAdmin):

list\_display = ('title', 'author', 'display\_genre')

inlines = [BooksInstanceInline]

Copy to Clipboard

Agora navegue até uma view pala um Book no seu site - na parte inferior, você verá as instâncias do livro relacionadas a este livro (imediatamente abaixo dos campos de gênero do livro):



Nesse caso, tudo o que fizemos foi declarar nossa classe inline tabular, que apenas adiciona todos os campos do modelo embutido. Você pode especificar todos os tipos de informações adicionais para o layout, incluindo os campos a serem exibidos, sua ordem, se eles são somente leitura ou não, etc. (veja [TabularInline](https://docs.djangoproject.com/en/dev/ref/contrib/admin/" \l "django.contrib.admin.TabularInline) para maiores informações).

Nota: Existem alguns limites dolorosos nesta funcionalidade! Na captura de tela acima, temos três instâncias de livros existentes, seguidas de três espaços reservados para novas instâncias de livros (que são muito semelhantes!). Seria melhor não ter instâncias do livro reserva por padrão e apenas adicioná-las com o link **Add another Book instance** , ou poder listar apenas BookInstances como links não legíveis daqui. A primeira opção pode ser feita configurando atributo extra para 0 no BooksInstanceInline model, tente você mesmo.

## [Desafie-se](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#desafie-se)

Aprendemos muito nesta seção, então agora é hora de você tentar algumas coisas.

1. Para a listview BookInstance, adicione o código para exibir o livro, o status, a data de devolução e o id (em vez do texto padrão \_\_str\_\_()).
2. Adicione uma listagem inline de itens Book para a lista detalhada de Author usando a mesma abordagem que fizemos para  Book/BookInstance.

## [Resumo](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#resumo)

É isso aí! Agora você aprendeu como configurar o site de administração na sua forma simples e aprimorada, como criar um superusuário, como navegar no site de administração e visualizar, excluir e atualizar registros. Ao longo do caminho você criou um monte de Livros, Instâncias de livros, Gêneros e Autores que poderemos listar e exibir assim que criarmos nossas próprias views e templates.

## [Leitura adicional](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site#leitura_adicional)

* [Escrevendo seu primeiro app Django, parte 2: Apresentando o Django Admin](https://docs.djangoproject.com/en/2.0/intro/tutorial02/#introducing-the-django-admin)  (Django docs)
* [O Django Admin site](https://docs.djangoproject.com/en/2.0/ref/contrib/admin/) (Django Docs)

# Django Tutorial Parte 5: Criando nossa home page

* [Anterior](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Admin_site)
* [Menu: Django](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django)
* [Próxima](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views)

Agora estamos prontos para adicionar o código que exibe nossa primeira página completa - uma home page do site [LocalLibrary](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial_local_library_website). A página inicial mostrará o número de registros que temos para cada tipo de modelo e fornecerá links de navegação na barra lateral para nossas outras páginas. Ao longo do caminho, obteremos experiência prática ao escrever mapas e visualizações básicos de URL, obter registros do banco de dados e usar modelos.

|  |  |
| --- | --- |
| **Pré-requisitos:** | Leia a [Introdução ao Django](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction). Conclua os tópicos do tutorial anterior (incluindo [Django Tutorial Part 4: Django admin site](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Admin_site)). |
| **Objetivo:** | Aprender a criar mapas e visualizações de URL simples (onde nenhum dado é codificado no URL), obtenha dados de modelos e crie modelos. |

## [Visão Global](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#vis%C3%A3o_global)

Depois de definirmos nossos modelos e criarmos alguns registros iniciais da biblioteca para trabalhar, é hora de escrever o código que apresenta essas informações aos usuários. A primeira coisa que precisamos fazer é determinar quais informações queremos exibir em nossas páginas e definir os URLs a serem usados para retornar esses recursos. Em seguida, criaremos um mapeador de URLs, visualizações e modelos para exibir as páginas.

O diagrama a seguir descreve o fluxo de dados principal e os componentes necessários ao manipular solicitações e respostas HTTP. Como já implementamos o modelo, os principais componentes que criaremos são:

* Mapeadores de URL para encaminhar os URLs suportados (e qualquer informação codificada nos URLs) para as funções de exibição apropriadas.
* View functions para obter os dados solicitados dos modelos, crie páginas HTML que exibem os dados e retorne as páginas ao usuário para visualização no navegador.
* Templates para usar ao renderizar dados nas visualizações.

Diagrama

Descrição gerada automaticamente

Como você verá na próxima seção, temos 5 páginas para exibir, o que é muita informação para documentar em um único artigo. Portanto, este artigo se concentrará em como implementar a página inicial e abordaremos as outras páginas em um artigo subsequente. Isso deve fornecer uma boa compreensão completa de como os mapeadores, visualizações e modelos de URL funcionam na prática.

## [Definindo os URLs do recurso](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#definindo_os_urls_do_recurso)

Como esta versão do [LocalLibrary](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial_local_library_website)é essencialmente somente leitura para usuários finais, precisamos fornecer uma página de destino para o site (uma página inicial) e páginas que exibam visualizações de lista e detalhes de livros e autores.

As URLs que iremos precisar na nossa página são:

* catalog/ — A página inicial (index).
* catalog/books/ — Uma lista de todos os livros.
* catalog/authors/ — Uma lista de todos os autores.
* catalog/book/<id> — A exibição de detalhes de um livro específico, com uma chave primária de campo <id> (o padrão). Por exemplo, o URL do terceiro livro adicionado à lista será /catalog/book/3.
* catalog/author/<id> — A exibição de detalhes para o autor específico com um campo de chave primária de *<id>.*Por exemplo, o URL do 11º autor adicionado à lista será  /catalog/author/11.

Os três primeiros URLs retornarão a página de índice, a lista de livros e a lista de autores. Esses URLs não codificam nenhuma informação adicional e as consultas que buscam dados no banco de dados sempre serão as mesmas. No entanto, os resultados retornados pelas consultas dependerão do conteúdo do banco de dados.

Por outro lado, os dois URLs finais exibirão informações detalhadas sobre um livro ou autor específico. Esses URLs codificam a identidade do item a ser exibido (representado por <id> acima). O mapeador de URLs extrairá as informações codificadas e as passará para a visualização, e a visualização determinará dinamicamente quais informações serão obtidas do banco de dados. Ao codificar as informações no URL, usaremos um único conjunto de mapeamento de URL, uma visualização e um modelo para lidar com todos os livros (ou autores).

**Nota**: Com o Django, você pode construir suas URLs da maneira que desejar - você pode codificar informações no corpo da URL, como mostrado acima, ou incluir GET parâmetros no URL, por exemplo /book/?id=6. Qualquer que seja a abordagem usada, os URLs devem ser mantidos limpos, lógicos e legíveis, conforme [recomendado pelo W3C](https://www.w3.org/Provider/Style/URI).  
A documentação do Django recomenda informações de codificação no corpo da URL para obter um melhor design da URL.

Conforme mencionado na visão geral, o restante deste artigo descreve como construir a página index.

## [Criando a página index](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#criando_a_p%C3%A1gina_index)

A primeira página que criaremos é a página index (catalog/). A página index incluirá algum HTML estático, juntamente com "contagens" geradas de diferentes registros no banco de dados. Para fazer isso funcionar, criaremos um mapeamento de URL, uma visualização e um modelo.

**Nota**: Vale a pena prestar um pouco de atenção extra nesta seção. A maioria das informações também se aplica às outras páginas que criaremos.

### [Mapeamento de URL](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#mapeamento_de_url)

Quando criamos o [esqueleto do website](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/skeleton_website), atualizamos o arquivo **locallibrary/urls.py** para garantir que sempre que um URL que comece com catalog/  é recebido, o módulo URLConf catalog.urls processará a substring restante.

O seguinte snippet de código de **locallibrary/urls.py**inclui o modulo catalog.urls:

urlpatterns += [

    path('catalog/', include('catalog.urls')),

]

**Nota**: Sempre que o Django encontra a função de importação  [django.urls.include()](https://docs.djangoproject.com/en/2.1/ref/urls/" \l "django.urls.include" \o "django.conf.urls.include), divide a string da URL no caractere final designado e envia a subsequência restante para o módulo URLconf incluído para processamento adicional.

Também criamos um arquivo de espaço reservado para o modulo URLConf, chamado **/catalog/urls.py**. Adicione as seguintes linhas a esse arquivo:

urlpatterns = [

path('', views.index, name='index'),

]

Copy to Clipboard

A função path() define o seguinte:

* Um padrão de URL, que é uma sequência vazia: ''. Discutiremos detalhadamente os padrões de URL ao trabalhar em outras visualizações.
* A view function that will be called if the URL pattern is detected: views.index,  which is the function named index() in the **views.py**file.

A função path() também especifica um parâmetro de nome, que é um identificador exclusivo para esse mapeamento de URL específico. Você pode usar o nome para "reverter" o mapeador, ou seja, para criar dinamicamente um URL que aponte para o recurso que o mapeador foi projetado para manipular. Por exemplo, podemos usar o parâmetro name para vincular à nossa home page a partir de qualquer outra página adicionando o seguinte link em um modelo:

<a href="{% url 'index' %}">Home</a>.

Copy to Clipboard

**Nota**: Podemos codificar o link como em <a href="**/catalog/**">Home</a>), mas se alterarmos o padrão da nossa página inicial, por exemplo, para /catalog/index) os modelos não serão mais vinculados corretamente. Usar um mapeamento de URL invertido é muito mais flexível e robusto.

### [View (function-based)](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#view_function-based)

Uma view é uma função que processa uma solicitação HTTP, busca os dados necessários no banco de dados, renderiza os dados em uma página HTML usando um modelo HTML e, em seguida, retorna o HTML gerado em uma resposta HTTP para exibir a página ao usuário. A visualização do índice segue esse modelo - ele busca informações sobre o número de Book, BookInstance, disponibilidade de  BookInstance e registros de Author que temos no banco de dados e passa essas informações para um modelo para exibição.

Abra **catalog/views.py** e observe que o arquivo já importa o [render()](https://docs.djangoproject.com/en/2.1/topics/http/shortcuts/#django.shortcuts.render) da função shortcuts para gerar arquivo HTML usando um modelo e dados:

from django.shortcuts import render

# Create your views here.

Copy to Clipboard

Cole as seguintes linhas na parte inferior do arquivo:

from catalog.models import Book, Author, BookInstance, Genre

def index(request):

"""View function for home page of site."""

# Generate counts of some of the main objects

num\_books = Book.objects.all().count()

num\_instances = BookInstance.objects.all().count()

# Available books (status = 'a')

num\_instances\_available = BookInstance.objects.filter(status\_\_exact='a').count()

# The 'all()' is implied by default.

num\_authors = Author.objects.count()

context = {

'num\_books': num\_books,

'num\_instances': num\_instances,

'num\_instances\_available': num\_instances\_available,

'num\_authors': num\_authors,

}

# Render the HTML template index.html with the data in the context variable

return render(request, 'index.html', context=context)

Copy to Clipboard

A primeira linha importa as classes de models que usaremos para acessar dados em todas as nossas visualizações.

A primeira parte da função view busca o número de registros usando o atributo  objects.all() nas classes de modelo. Também recebe uma lista de objetos de  BookInstance que possuem um valor de 'a' (Disponibilidade) no campo status. Você pode encontrar mais informações sobre como acessar os dados do modelo em nosso tutorial anterior [Django Tutorial Part 3: Using models > Searching for records](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Models#searching_for_records).

No final da função view chamamos a função render() para criar uma página HTML e retornar a página como resposta. essa função de atalho envolve várias outras funções para simplificar um caso de uso muito comum. A função render() aceita os seguintes parâmetros:

* o objeto request original, que é um HttpRequest.
* um modelo HTML com espaços reservados para os dados.
* uma variável context, que é um dicionário Python, contendo os dados a serem inseridos nos espaços reservados.

Falaremos mais sobre modelos e variáveis context na próxima seção. Vamos criar nosso template para que possamos exibir algo para o usuário!

### [Template](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#template)

Um template é um arquivo de texto que define a estrutura ou o layout de um arquivo (como uma página HTML), usa espaços reservados para representar o conteúdo real.

Django irá procurar automaticamente templates na pasta chamada '**templates**' em sua aplicação. Por exemplo, na exibição de index que acabamos de adicionar, a função render() espera encontrar o arquivo ***index.html*** em **/locallibrary/catalog/templates/** e gera um erro se o arquivo não estiver presente.

Você pode verificar isso salvando as alterações anteriores e acessando 127.0.0.1:8000 no seu navegador - ele exibirá uma mensagem de erro bastante intuitiva: "TemplateDoesNotExist at /catalog/", e outros detalhes.

**Nota**: Com base no arquivo de configurações do seu projeto, o Django procurará templates em vários locais, pesquisando nos aplicativos instalados por padrão. Você pode descobrir mais sobre como o Django encontra templates e quais formatos ele suporta no [the Templates section of the Django documentation](https://docs.djangoproject.com/en/2.1/topics/templates/).

#### Estendendo templates

O index template precisará de marcação HTML padrão para head e a body, juntamente com as seções de navegação para criar um link para as outras páginas do site (que ainda não criamos) e para as seções que exibem dados introdutórios de texto e livro.

Grande parte da estrutura HTML e de navegação será a mesma em todas as páginas do nosso site. Em vez de duplicar o código padrão em todas as páginas, você pode usar a linguagem de modelagem do Django para declarar um modelo base e depois estendê-lo para substituir apenas os bits que são diferentes para cada página específica.

O seguinte snippet de código é um template base de amostra de um arquivo **base\_generic.html**. Em breve, criaremos o modelo para a LocalLibrary. O exemplo abaixo inclui HTML comum com seções para um título, uma barra lateral e o conteúdo principal marcado com as template tags de nome block e endblock, mostrado em negrito. Você pode deixar os blocos vazios ou incluir o conteúdo padrão a ser usado ao renderizar páginas derivadas do modelo.

**Nota**: Template tags são funções que você pode usar em um modelo para percorrer as listas, executar operações condicionais com base no valor de uma variável e assim por diante. Além das template tags, a sintaxe template permite que você faça referência a variáveis que são passadas para a template a partir da view e use filtros de template para formatar variáveis (por exemplo, para converter uma sequência em minúscula).

<!DOCTYPE html>

<html lang="en">

<head>

{% block title %}<title>Local Library</title>{% endblock %}

</head>

<body>

{% block sidebar %}<!-- insert default navigation text for every page -->{% endblock %}

{% block content %}<!-- default content text (typically empty) -->{% endblock %}

</body>

</html>

Copy to Clipboard

Ao definir um template para uma visualização específica, primeiro especificamos o template base usando a template tag extends — veja o exemplo de código abaixo. Em seguida, declaramos quais seções do template queremos substituir (se houver), usando seções block/endblock como no template base.

Por exemplo, o trecho de código abaixo mostra como usar a template tag extends e substituir o block content. O HTML gerado incluirá o código e a estrutura definidos no template base, incluindo o conteúdo padrão que você definiu no block title, mas o novo block content no lugar do padrão.

{% extends "base\_generic.html" %}

{% block content %}

<h1>Local Library Home</h1>

<p>Welcome to LocalLibrary, a website developed by <em>Mozilla Developer Network</em>!</p>

{% endblock %}

Copy to Clipboard

#### O template base LocalLibrary

Usaremos o seguinte snippet de código como modelo básico para o site LocalLibrary. Como você pode ver, ele contém algum código HTML e define blocos para title, sidebar, e content. Temos um título padrão e uma barra lateral padrão com links para listas de todos os livros e autores, ambos colocados em blocos para serem facilmente alterados no futuro.

**Nota**: Também introduzimos duas template tags adicionais: url e load static. Essas tags serão explicadas nas próximas seções.

Crie um novo arquivo **base\_generic.html**em **/locallibrary/catalog/templates/** e cole o seguinte código no arquivo:

<!DOCTYPE html>

<html lang="en">

<head>

{% block title %}<title>Local Library</title>{% endblock %}

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

<!-- Add additional CSS in static file -->

{% load static %}

<link rel="stylesheet" href="{% static 'css/styles.css' %}">

</head>

<body>

<div class="container-fluid">

<div class="row">

<div class="col-sm-2">

{% block sidebar %}

<ul class="sidebar-nav">

<li><a href="{% url 'index' %}">Home</a></li>

<li><a href="">All books</a></li>

<li><a href="">All authors</a></li>

</ul>

{% endblock %}

</div>

<div class="col-sm-10 ">{% block content %}{% endblock %}</div>

</div>

</div>

</body>

</html>

Copy to Clipboard

O template inclui CSS de [Bootstrap](http://getbootstrap.com/) para melhorar o layout e a apresentação da página HTML. O uso do Bootstrap (ou outra estrutura da Web do lado do cliente) é uma maneira rápida de criar uma página atraente que é exibida bem em diferentes tamanhos de tela.

O template base também faz referência a um arquivo css local (**styles.css**) que fornece estilo adicional. Criar um arquivo **styles.css**em**/locallibrary/catalog/static/css/** e cole o seguinte código no arquivo:

.sidebar-nav {

margin-top: 20px;

padding: 0;

list-style: none;

}

Copy to Clipboard

#### O template index

Crie um novo arquivo HTML **index.html**em **/locallibrary/catalog/templates/** e cole o seguinte código no arquivo Esse código estende nosso modelo base na primeira linha e substitui o padrão block content para o template.

{% extends "base\_generic.html" %}

{% block content %}

<h1>Local Library Home</h1>

<p>Welcome to LocalLibrary, a website developed by <em>Mozilla Developer Network</em>!</p>

<h2>Dynamic content</h2>

<p>The library has the following record counts:</p>

<ul>

<li><strong>Books:</strong> {{ num\_books }}</li>

<li><strong>Copies:</strong> {{ num\_instances }}</li>

<li><strong>Copies available:</strong> {{ num\_instances\_available }}</li>

<li><strong>Authors:</strong> {{ num\_authors }}</li>

</ul>

{% endblock %}

Copy to Clipboard

Na seção Dynamic content, declaramos espaços reservados (variáveis de template) para as informações da exibição que queremos incluir. As variáveis são colocadas entre chaves (guiador), como mostrado em negrito no exemplo de código.

**Nota:** Você pode reconhecer facilmente variáveis de template e template tags (funções) - as variáveis são colocadas entre chaves ({{ num\_books }}), e as tags são colocadas em chaves simples com sinais de porcentagem ({% extends "base\_generic.html" %}).

O importante a ser observado aqui é que as variáveis são nomeadas com as chaves que passamos para o dicionário context na função render() da nossa view (veja a amostra abaixo). As variáveis serão substituídas pelos valores associados quando o modelo for renderizado.

context = {

'num\_books': num\_books,

'num\_instances': num\_instances,

'num\_instances\_available': num\_instances\_available,

'num\_authors': num\_authors,

}

return render(request, 'index.html', context=context)

Copy to Clipboard

#### Referenciando arquivos estáticos nos templates

É provável que seu projeto use recursos estáticos, incluindo JavaScript, CSS e imagens. Como a localização desses arquivos pode não ser conhecida (ou pode mudar), o Django permite que você especifique a localização em seus modelos em relação a configuração global STATIC\_URL. O site padrão do esqueleto define o valor de STATIC\_URL para '/static/', mas você pode optar por hospedá-los em uma rede de entrega de conteúdo ou em outro local.

Dentro do template que você chama primeiro na template tag load especificando "static" para adicionar a biblioteca de modelos, conforme mostrado no exemplo de código abaixo. Você pode então usar a template tag static e especifique o URL relativo ao arquivo necessário.

<!-- Add additional CSS in static file -->

{% load static %}

<link rel="stylesheet" href="{% static 'css/styles.css' %}">

Copy to Clipboard

Você pode adicionar uma imagem à página de maneira semelhante, por exemplo:

{% load static %}

<img src="{% static 'catalog/images/local\_library\_model\_uml.png' %}" alt="UML diagram" style="width:555px;height:540px;">

Copy to Clipboard

**Nota**: Os exemplos acima especificam onde os arquivos estão localizados, mas o Django não os serve por padrão. Configuramos o servidor da web de desenvolvimento para exibir arquivos modificando o mapeador de URL global (**/locallibrary/locallibrary/urls.py**) quando [criamos o esqueleto do website](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/skeleton_website), mas ainda precisamos ativar a veiculação de arquivos na produção. Veremos isso mais tarde.

Para obter mais informações sobre como [Trabalhar com arquivos estaticos](https://docs.djangoproject.com/en/2.1/howto/static-files/), consulte Gerenciando arquivos estáticos na documentação do Django.

#### Vinculando as URLs

O template base abaixo introduziu a template tag url.

<li><a href="{% url 'index' %}">Home</a></li>

Copy to Clipboard

Essa tag aceita o nome de uma função path() chamado em **urls.py** e os valores para quaisquer argumentos que a view associada receberá dessa função e retorna um URL que você pode usar para vincular ao recurso.

#### Configurando onde encontrar os templates

Você precisa dizer ao Django para procurar seus templates na pasta de templates. Para fazer isso, adicione o diretório de templates ao objeto TEMPLATES editando o arquivo **settings.py**, como mostrado em negrito, no seguinte exemplo de código:

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [

os.path.join(BASE\_DIR, 'templates'),

],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

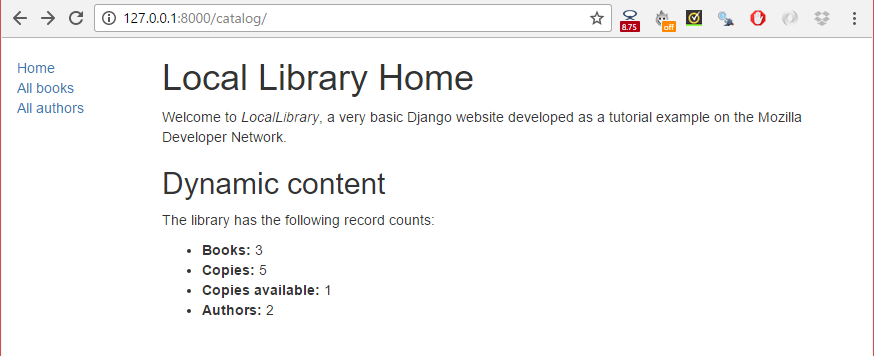
},

]

Copy to Clipboard

## [Com o que se parece?](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#com_o_que_se_parece)

Neste ponto, criamos todos os recursos necessários para exibir a página index. Execute o servidor (python3 manage.py runserver) e abra <http://127.0.0.1:8000/> no seu navegador. Se tudo estiver configurado corretamente, seu site deverá ter a seguinte captura de tela.



**Nota:** Os links **All books** e **All authors** ainda não funcionarão porque os caminhos, visualizações e modelos para essas páginas não estão definidos. Acabamos de inserir espaços reservados para esses links no template base\_generic.html.

## [Desafie-se](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#desafie-se)

Temos duas tarefas para testar a sua familiaridade com as consultas de modelos, views e templates

1. O modelo de [base](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#The_LocalLibrary_base_template) da BibliotecaLocal inclui um bloco de título. Substitua este bloco no modelo de índice e crie um novo título para a página.
2. **Dica:**A seção[Extendendo Templates](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#extending_templates) explica como criar blocos e  extender um bloco em outro template.
3. Modifique a [view](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#view_(function-based)) para gerar contagens para gêneros e livros que contenham uma palavra específica (case insensitive), e passe o resultado para o contexto. Isso é feito de maneira semelhante à criação e uso de num\_books e num\_instances\_available. Em seguida, atualize o template do index para incluir essas variáveis.

## [Resumo](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#resumo)

Acabamos de criar a página inicial do nosso site - uma página HTML que exibe uma série de registros do banco de dados e links para outras páginas ainda a serem criadas. Ao longo do caminho, aprendemos informações fundamentais sobre mapeadores de url, views, consulta do banco de dados com modelos, passagem de informações para um modelo a partir de uma view e criação e extensão de templates.

No próximo artigo, continuaremos sobre esse conhecimento para criar as quatro páginas restantes de nosso site.

## [Veja também](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page#veja_tamb%C3%A9m)

* [Escrevendo sua primeira aplicação Django, parte 3: View e Templates](https://docs.djangoproject.com/pt-br/3.1/intro/tutorial03/) (documentação do Django)
* [Despachante de URL](https://docs.djangoproject.com/pt-br/3.1/topics/http/urls/) (Django docs)
* [Funções das Views](https://docs.djangoproject.com/pt-br/3.1/topics/http/views/) (DJango docs)
* [Templates](https://docs.djangoproject.com/pt-br/3.1/topics/templates/) (Django docs)
* [Gerenciando arquivos estáticos](https://docs.djangoproject.com/en/2.1/howto/static-files/) (Django docs)
* [Funções de atalho do Django](https://docs.djangoproject.com/en/2.1/topics/http/shortcuts/#django.shortcuts.render) (Django docs)

# Tutorial Django Parte 6: Lista genérica e detail views

* [Anterior](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Home_page)
* [Menu: Django](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django)
* [Próxima](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Sessions)

Este tutorial estende nosso website [LocalLibrary](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial_local_library_website), 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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#vis%C3%A3o_global)

Neste tutorial, vamos concluir a primeira versão do website [LocalLibrary](https://developer.mozilla.org/pt-br/docs/Learn/Server-side/Django/Tutorial_local_library_website) 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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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'),

]

Copy to Clipboard

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)](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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

Copy to Clipboard

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 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

Copy to Clipboard

#### 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 books containing the title war

Copy to Clipboard

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

Copy to Clipboard

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, retorne o novo contexto (atualizado).

**Nota**: Confira [Built-in class-based generic views](https://docs.djangoproject.com/pt-br/2.1/topics/class-based-views/generic-display/) (Django docs) para muitos mais exemplos do que você pode fazer.

### [Criando o template List View](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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> ({{book.author}})

</li>

{% endfor %}

</ul>

{% else %}

<p>There are no books in the library.</p>

{% endif %}

{% endblock %}

Copy to Clipboard

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](https://docs.djangoproject.com/en/2.1/ref/templates/builtins/" \l "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 %}

Copy to Clipboard

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](https://docs.djangoproject.com/pt-br/2.1/ref/templates/builtins/" \l "if), [ifequal/ifnotequal](https://docs.djangoproject.com/pt-br/2.1/ref/templates/builtins/" \l "ifequal-and-ifnotequal), e [ifchanged](https://docs.djangoproject.com/pt-br/2.1/ref/templates/builtins/" \l "ifchanged) em [Built-in template tags and filters](https://docs.djangoproject.com/pt-br/2.1/ref/templates/builtins) (Django Docs).

#### For loops

O template usa as template tags [for](https://docs.djangoproject.com/pt-br/2.1/ref/templates/builtins/#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 %}

Copy to Clipboard

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}})

Copy to Clipboard

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>

Copy to Clipboard

### [Com o que se parece?](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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 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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#url_mapping_2)

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-detail'),

]

Copy to Clipboard

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](https://docs.djangoproject.com/pt-br/2.1/topics/http/urls/#path-converters) 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 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()](https://docs.djangoproject.com/en/2.1/ref/urls/" \l "django.urls.re_path) method.

This method is used just like path() except that it allows you to specify a pattern using a [Regular expression](https://docs.python.org/3/library/re.html). For example, the previous path could have been written as shown below:

re\_path(r'^book/(?P<pk>\d+)$', views.BookDetailView.as\_view(), name='book-detail'),

Copy to Clipboard

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 (\_) |
| + | 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** |
| --- | --- |
| **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. |

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](https://docs.djangoproject.com/pt-br/2.1/topics/http/urls/#views-extra-options) 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'}, name='aurl'),

path('anotherurl/', views.my\_reused\_view, {'my\_template\_name': 'another\_path'}, name='anotherurl'),

Copy to Clipboard

**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)](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#view_class-based_2)

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

class BookDetailView(generic.DetailView):

model = Book

Copy to Clipboard

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': book})

Copy to Clipboard

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': book})

Copy to Clipboard

### [Criando o template Detail View](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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> <!-- 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 %}

Copy to Clipboard

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 %}

Copy to Clipboard

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: UnorderedObjectListWarning: Pagination may yield inconsistent results with an unordered object\_list: <QuerySet [<Author: Ortiz, David>, <Author: H. McRaven, William>, <Author: Leigh, Melinda>]>

allow\_empty\_first\_page=allow\_empty\_first\_page, \*\*kwargs)

Isso acontece porque o [objeto paginator](https://docs.djangoproject.com/pt-br/2.1/topics/pagination/#paginator-objects) 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.

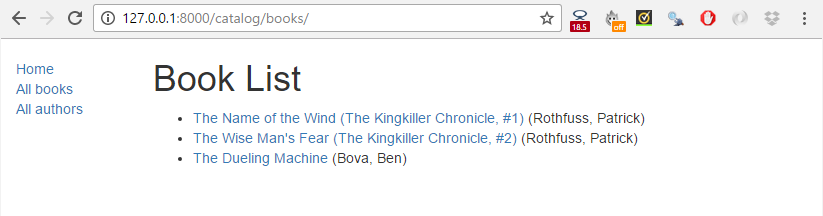
E por último, mas não menos importante, você deve classificar por um atributo/coluna que realmente tenha um índice (exclusivo ou não) em seu banco de dados para evitar problemas de desempenho. Obviamente, isso não será necessário aqui (e provavelmente estamos nos adiantando muito) com tão poucos livros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

## [Com o que se parece agora?](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#com_o_que_se_parece_agora)

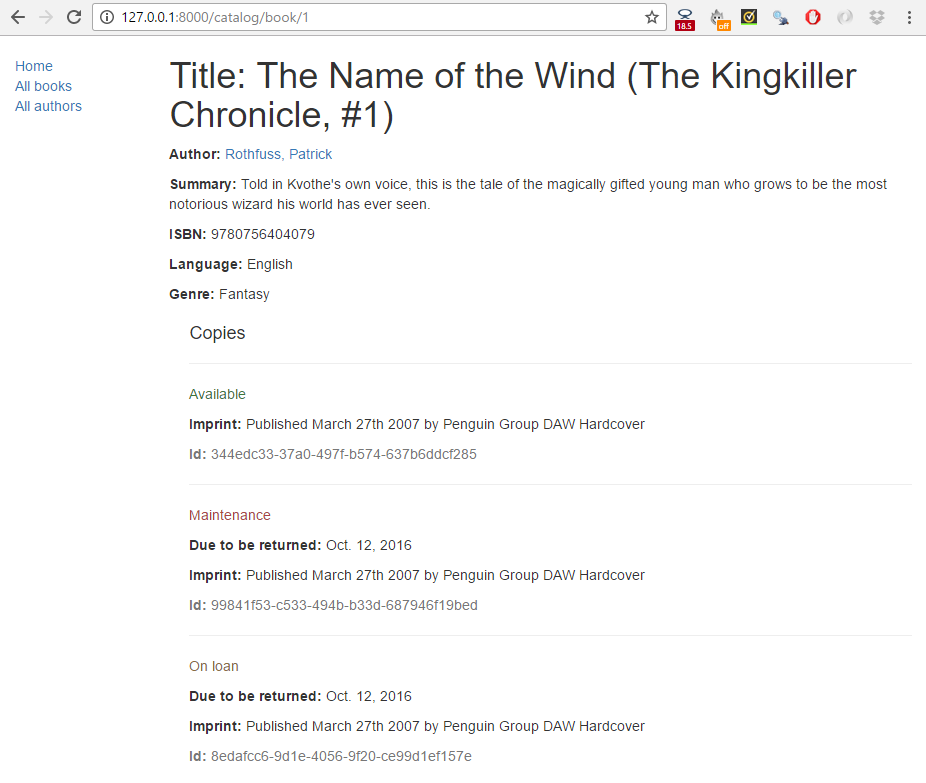
Nesse ponto, deveríamos ter criado tudo o necessário para exibir a lista de livros e as páginas de detalhes do livro. Execute o servidor (python3 manage.py runserver) e abra no seu navegador <http://127.0.0.1:8000/>.

**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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#pagina%C3%A7%C3%A3o)

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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#views)

Abra **catalog/views.py**, e adicionea linha paginate\_by mostrado em negrito abaixo.

class BookListView(generic.ListView):

model = Book

paginate\_by = 10

Copy to Clipboard

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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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.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 %}

Copy to Clipboard

O page\_obj é um objeto de [Paginator](https://docs.djangoproject.com/pt-br/2.1/topics/pagination/" \l "paginator-objects) 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?](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#com_o_que_se_parece_agora_2)

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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views" \l "update_the_base_template) 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](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#creating_the_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><strong>Author:</strong> <a href="{% url 'author-detail' book.author.pk %}">{{ book.author }}</a></p>

Copy to Clipboard

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





## [Summary](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Generic_views#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 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.

# Django Tutorial Parte 8: Autenticação e permissões do usuário

* [Anterior](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Sessions)
* [Visão geral: Django](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django)
* [Próximo](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Forms)

Neste tutorial, mostraremos como permitir que os usuários façam login em seu site com suas próprias contas e como controlar o que podem fazer e ver com base em se estão ou não conectados e suas permissões. Como parte desta demonstração, estenderemos o site [locallibrary](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial_local_library_website), adicionando páginas de login e logout, e páginas específicas para o usuário e funcionários para visualização de livros que foram emprestados.

|  |  |
| --- | --- |
| **Pré-requisitos:** | Complete todos os tópicos do tutorial anterior, até e incluindo [Django Tutorial Parte 7: Quadro de sessões](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Sessions). |
| **Objetivo:** | Para entender como configurar e usar autenticação e permissões do usuário. |

## [Visão geral](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#overview)

O Django fornece um sistema de autenticação e autorização ("permissão"), construído em cima da estrutura de sessão discutida no [tutorial anterior](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Sessions), que permite verificar as credenciais do usuário e definir quais ações cada usuário pode realizar. A estrutura inclui modelos Users e Groups (uma maneira genérica de aplicar permissões a mais de um usuário de cada vez), permissões/bandeiras que designam se um usuário pode executar uma tarefa, formulários e visualizações para fazer login nos usuários e exibir ferramentas para restringir conteúdo.

**Nota:** De acordo com Django, o sistema de autenticação visa ser muito genérico, e por isso não fornece alguns recursos fornecidos em outros sistemas de autenticação web. Soluções para alguns problemas comuns estão disponíveis como pacotes de terceiros. Por exemplo, estrangulamento de tentativas de login e autenticação contra terceiros (por exemplo, OAuth).

Neste tutorial, mostraremos como ativar a autenticação do usuário no site [localLibrary](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial_local_library_website), criar suas próprias páginas de login e logout, adicionar permissões aos seus modelos e controlar o acesso às páginas. Usaremos a autenticação/permissões para exibir listas de livros que foram emprestados para usuários e bibliotecários.

O sistema de autenticação é muito flexível, e você pode construir seus URLs, formulários, visualizações e modelos do zero, se quiser, apenas chamando a API fornecida para fazer login no usuário. No entanto, neste artigo, vamos usar as visualizações de autenticação "stock" da Django e formulários para nossas páginas de login e logout. Ainda precisamos criar alguns modelos, mas isso é muito fácil.

Também mostraremos como criar permissões e verificaremos o status de login e as permissões em visualizações e modelos.

## [Habilitando a autenticação](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#enabling_authentication)

A autenticação foi ativada automaticamente quando [criamos o site esqueleto](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/skeleton_website) (no tutorial 2) para que você não precise fazer mais nada neste momento.

**Nota:** A configuração necessária foi feita para nós quando criamos o aplicativo usando o comando django-admin startproject. As tabelas de banco de dados para usuários e permissões de modelos foram criadas quando chamamos a python manage.py migrate.

A configuração está configurada nas seções INSTALLED\_APPS e MIDDLEWARE do arquivo do projeto (**locallibrary/locallibrary/configurações.py**), conforme mostrado abaixo:

INSTALLED\_APPS = [

...

'django.contrib.auth', #Core authentication framework and its default models.

'django.contrib.contenttypes', #Django content type system (allows permissions to be associated with models).

....

MIDDLEWARE = [

...

'django.contrib.sessions.middleware.SessionMiddleware', #Manages sessions across requests

...

'django.contrib.auth.middleware.AuthenticationMiddleware', #Associates users with requests using sessions.

....

Copiar para área de transferência

## [Criando usuários e grupos](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#creating_users_and_groups)

Você já criou seu primeiro usuário quando olhamos para o [site de administração Django](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Admin_site) no tutorial 4 (este era um superusuário, criado com o comando python manage.py createsuperuser). Nosso superusuário já é autenticado e tem todas as permissões, por isso precisaremos criar um usuário de teste para representar um usuário normal do site. Usaremos o site administrativo para criar nossos grupos de mineração local e logins de sites, pois é uma das maneiras mais rápidas de fazê-lo.

**Nota:** Você também pode criar usuários programáticamente, como mostrado abaixo. Você teria que fazer isso, por exemplo, se desenvolver uma interface para permitir que usuários "comuns" criem seus próprios logins (você não deve dar à maioria dos usuários acesso ao site do administrador).

from django.contrib.auth.models import User

# Create user and save to the database

user = User.objects.create\_user('myusername', 'myemail@crazymail.com', 'mypassword')

# Update fields and then save again

user.first\_name = 'John'

user.last\_name = 'Citizen'

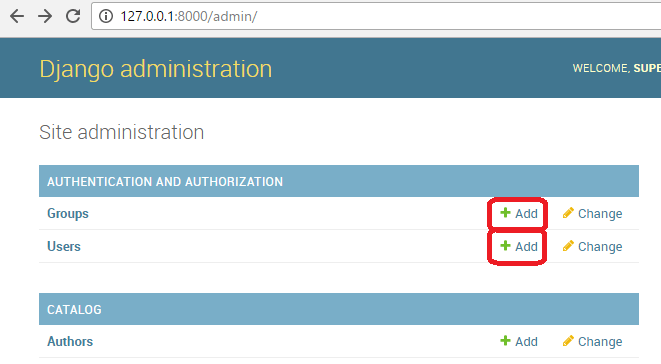
user.save()

Copiar para área de transferência

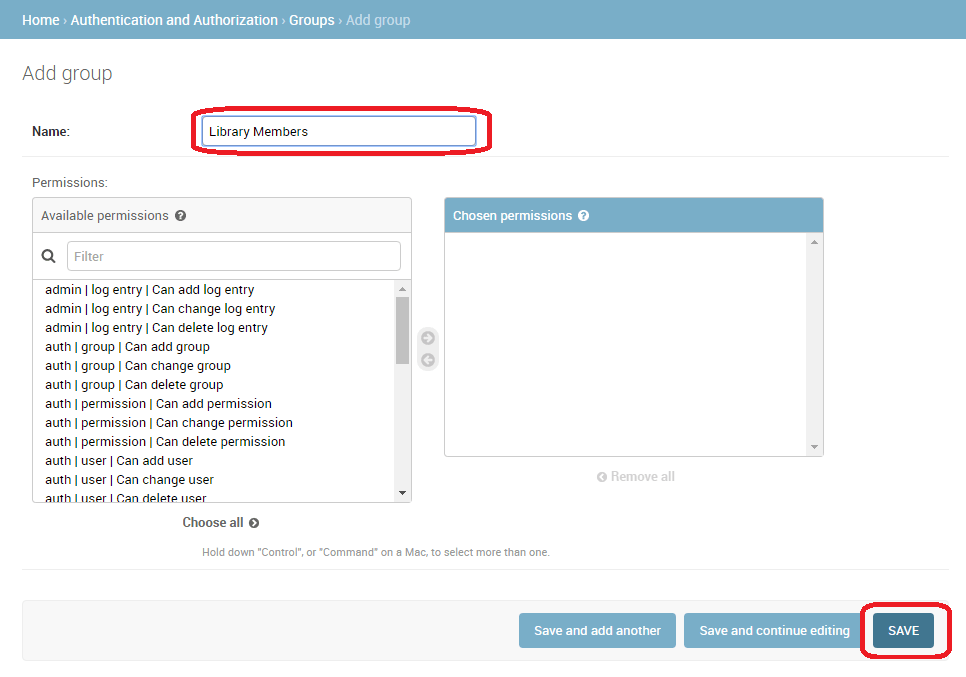
É altamente recomendável configurar um modelo de usuário personalizado ao iniciar um projeto real. Você poderá personalizá-lo facilmente no futuro se surgir a necessidade. Para obter mais informações, consulte [Usar um modelo de usuário personalizado ao iniciar um projeto](https://docs.djangoproject.com/en/4.0/topics/auth/customizing/#using-a-custom-user-model-when-starting-a-project) (docs Django).

Abaixo vamos primeiro criar um grupo e depois um usuário. Mesmo que ainda não tenhamos permissões para adicionar aos membros da biblioteca, se precisarmos mais tarde, será muito mais fácil adicioná-los uma vez ao grupo do que individualmente a cada membro.

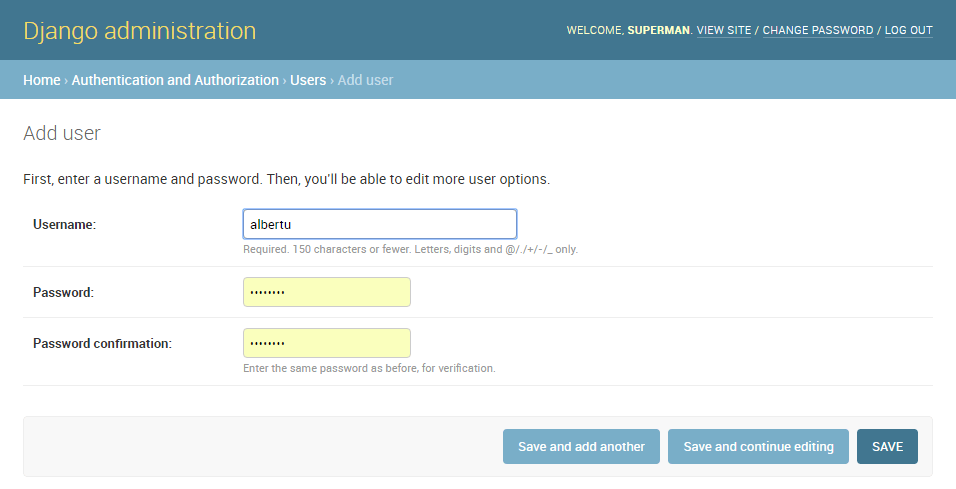
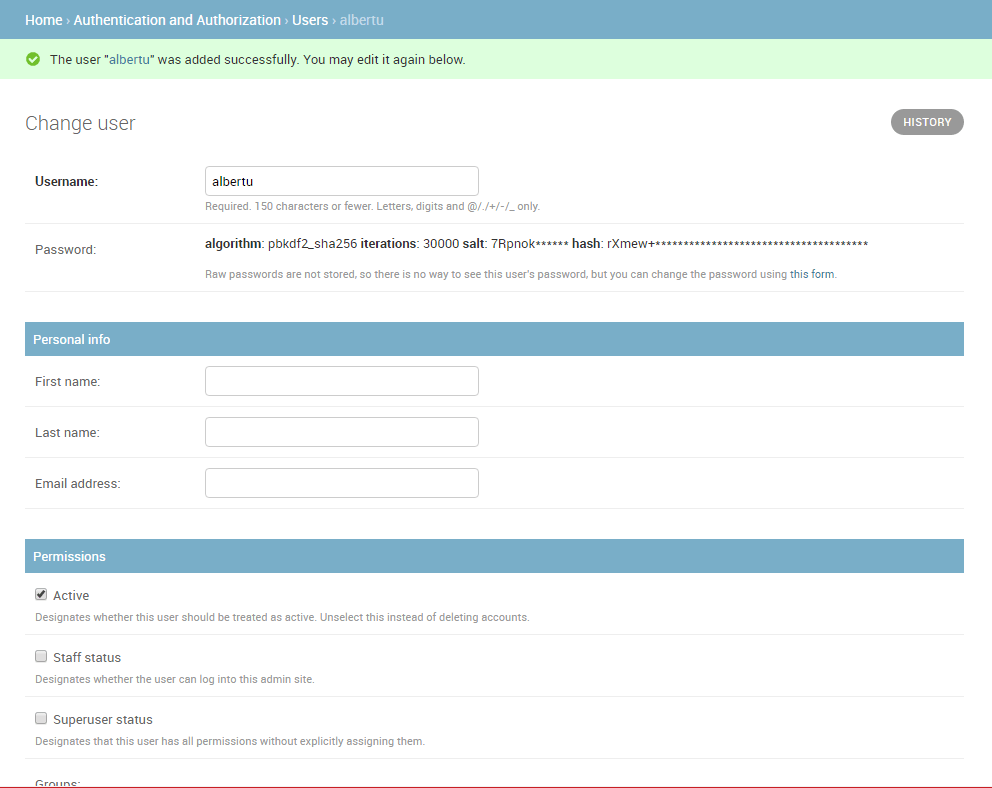
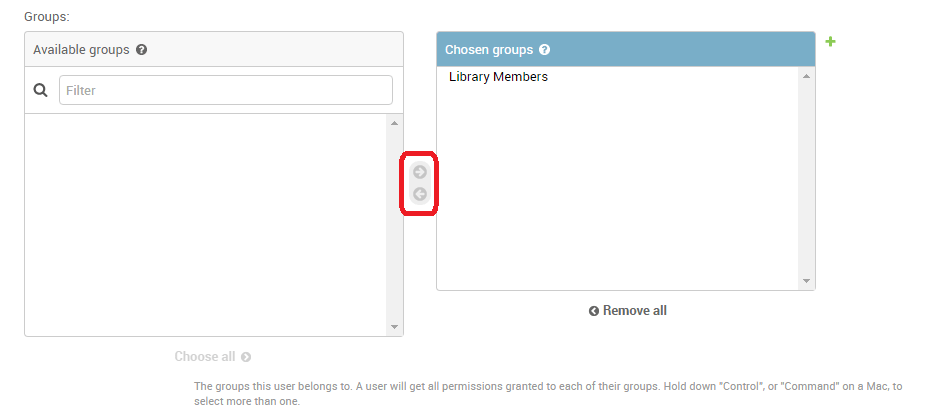
Inicie o servidor de desenvolvimento e navegue até o site administrativo no seu navegador local (http://127.0.0.1:8000/admin/). Faça login no site usando as credenciais para sua conta de superusuário. O nível superior do site do Administrador exibe todos os seus modelos, classificados por "aplicativo Django". A partir da seção **Autenticação e Autorização**, você pode clicar nos links **usuários** ou **grupos** para ver seus registros existentes.



Primeiro vamos criar um novo grupo para nossos membros da biblioteca.

1. Clique no botão **Adicionar** (ao lado do Grupo) para criar um novo Grupo; digite o **nome** "Membros da Biblioteca" para o grupo.
2. Não precisamos de permissões para o grupo, então basta pressionar **SAVE** (você será levado para uma lista de grupos).

Agora vamos criar um usuário:

1. Navegue de volta para a página inicial do site de administração
2. Clique no botão **Adicionar** ao lado dos usuários para abrir a caixa de diálogo Adicionar usuário.
3. Digite um **nome de usuário** apropriado e **confirmação de senha**/**senha** para o usuário de teste
4. Pressione **SAVE** para criar o usuário. O site de administração criará o novo usuário e imediatamente o levará a uma tela de usuário Change onde você pode alterar seu **nome de usuário** e adicionar informações para os campos opcionais do modelo do usuário. Esses campos incluem o primeiro nome, sobrenome, endereço de e-mail e o status e permissões do usuário (apenas a bandeira **Ativa** deve ser definida). Mais adiante, você pode especificar os grupos e permissões do usuário e ver datas importantes relacionadas ao usuário (por exemplo, sua data de adesão e última data de login).
5. Na seção Grupos, selecione o grupo **Membro da Biblioteca** na lista de grupos disponíveis e pressione a seta direita entre as caixas para **movê-la** para a caixa de grupos Escolhidos.
6. Não precisamos fazer mais nada aqui, então apenas selecione **SAVE** novamente, para ir à lista de usuários.

É isso mesmo! Agora você tem uma conta "membro normal da biblioteca" que você poderá usar para testes (uma vez que implementamos as páginas para permitir que eles façam login).

**Nota:** Você deve tentar criar outro usuário membro da biblioteca. Além disso, crie um grupo para bibliotecários, e adicione um usuário a isso também!

## [Configurando suas visualizações de autenticação](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#setting_up_your_authentication_views)

O Django fornece quase tudo o que você precisa para criar páginas de autenticação para lidar com o login, o log out e o gerenciamento de senhas "fora da caixa". Isso inclui um mapeador de URL, visualizações e formulários, mas não inclui os modelos — temos que criar os nossos!

Nesta seção, mostramos como integrar o sistema padrão no site localLibrary e criar os modelos. Vamos colocá-los no projeto principal URLs.

**Nota:** Você não tem que usar nenhum deste código, mas é provável que você vai querer porque torna as coisas muito mais fáceis. Você certamente precisará alterar o código de manipulação de formulários se você alterar seu modelo de usuário (um tópico avançado!) mas mesmo assim, você ainda seria capaz de usar as funções de exibição de estoque.

**Nota:** Neste caso, poderíamos razoavelmente colocar as páginas de autenticação, incluindo os URLs e modelos, dentro do nosso aplicativo de catálogo. No entanto, se tivéssemos vários aplicativos seria melhor separar esse comportamento de login compartilhado e tê-lo disponível em todo o site, então é isso que mostramos aqui!

### [Urls do Projeto](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#project_urls)

Adicione o seguinte ao fundo do arquivo urls.py do projeto (**locallibrary/locallibrary/urls.py**) arquivo:

#Add Django site authentication urls (for login, logout, password management)

urlpatterns += [

path('accounts/', include('django.contrib.auth.urls')),

]

Copiar para área de transferência

Navegue até a http://127.0.0.1:8000/accounts/ (observe a barra para a frente!). Django mostrará um erro que não conseguiu encontrar esta URL e listará todos os URLs que tentou. A partir disso você pode ver as URLs que funcionarão, por exemplo:

**Nota:** O uso do método acima adiciona os seguintes URLs com nomes em suportes quadrados, que podem ser usados para reverter os mapeamentos de URL. Você não precisa implementar mais nada — o mapeamento acima de URL mapeia automaticamente os URLs abaixo mencionados.

accounts/ login/ [name='login']

accounts/ logout/ [name='logout']

accounts/ password\_change/ [name='password\_change']

accounts/ password\_change/done/ [name='password\_change\_done']

accounts/ password\_reset/ [name='password\_reset']

accounts/ password\_reset/done/ [name='password\_reset\_done']

accounts/ reset/<uidb64>/<token>/ [name='password\_reset\_confirm']

accounts/ reset/done/ [name='password\_reset\_complete']

Copiar para área de transferência

Agora tente navegar até a URL de login (http://127.0.0.1:8000/accounts/login/). Isso falhará novamente, mas com um erro que diz que estamos perdendo o modelo necessário (**registration/login.html**) no caminho de pesquisa de modelo. Você verá as seguintes linhas listadas na seção amarela na parte superior:

Exception Type: TemplateDoesNotExist

Exception Value: registration/login.html

Copiar para área de transferência

O próximo passo é criar um diretório de registro no caminho de pesquisa e, em seguida, adicionar o **arquivo .html login**.

### [Diretório de modelos](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#template_directory)

Os URLs (e implicitamente, visualizações) que acabamos de adicionar esperam encontrar seus modelos associados em um diretório **/registro/** em algum lugar no caminho de pesquisa de modelos.

Para este site, colocaremos nossas páginas HTML nos **modelos/registro/**diretório. Este diretório deve estar no diretório raiz do seu projeto, ou seja, o mesmo diretório que o **catálogo** e pastas **de** laboratórios locais. Por favor, crie essas pastas agora.

**Nota:** Sua estrutura de pasta deve agora se parecer com a abaixo:

locallibrary/ #Django project folder

catalog/

locallibrary/

templates/

registration/

Para tornar o diretório **de modelos** visível para o carregador de modelo, precisamos adicioná-lo no caminho de pesquisa de modelo. Abra as configurações do projeto (**/locallibrary/locallibrary/configurações.py**).

Em seguida, importe o módulo os operacional (adicione a seguinte linha perto da parte superior do arquivo).

import os # needed by code below

Copiar para área de transferência

Atualize a linha 'DIRS' da seção TEMPLATES, conforme mostrado:

...

TEMPLATES = [

{

...

'DIRS': [os.path.join(BASE\_DIR, 'templates')],

'APP\_DIRS': True,

...

Copiar para área de transferência

### [Modelo de login](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#login_template)

**Aviso:** Os modelos de autenticação fornecidos neste artigo são uma versão muito básica/ligeiramente modificada dos modelos de login de demonstração do Django. Você pode precisar personalizá-los para seu próprio uso!

Crie um novo arquivo HTML chamado /**locallibrary/templates/registration/login.html** e dê-lhe o seguinte conteúdo:

{% extends "base\_generic.html" %}

{% block content %}

{% if form.errors %}

<p>Your username and password didn't match. Please try again.</p>

{% endif %}

{% if next %}

{% if user.is\_authenticated %}

<p>Your account doesn't have access to this page. To proceed,

please login with an account that has access.</p>

{% else %}

<p>Please login to see this page.</p>

{% endif %}

{% endif %}

<form method="post" action="{% url 'login' %}">

{% csrf\_token %}

<table>

<tr>

<td>{{ form.username.label\_tag }}</td>

<td>{{ form.username }}</td>

</tr>

<tr>

<td>{{ form.password.label\_tag }}</td>

<td>{{ form.password }}</td>

</tr>

</table>

<input type="submit" value="login" />

<input type="hidden" name="next" value="{{ next }}" />

</form>

{# Assumes you setup the password\_reset view in your URLconf #}

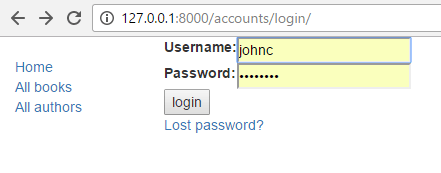
<p><a href="{% url 'password\_reset' %}">Lost password?</a></p>

{% endblock %}

Copiar para área de transferência

Este modelo compartilha algumas semelhanças com as que vimos antes — ele estende nosso modelo base e substitui o bloco de content. O resto do código é um código de manuseio de formulários bastante padrão, que discutiremos em um tutorial posterior. Tudo o que você precisa saber por enquanto é que isso exibirá um formulário no qual você pode inserir seu nome de usuário e senha, e que se você inserir valores inválidos, você será solicitado a inserir valores corretos quando a página atualizar.

Navegue de volta para a página de login (http://127.0.0.1:8000/accounts/login/) uma vez que você salvou seu modelo, e você deve ver algo assim:



Se você fizer login usando credenciais válidas, você será redirecionado para outra página (por padrão, isso será http://127.0.0.1:8000/accounts/profile/). O problema é que, por padrão, Django espera que ao fazer login você queira ser levado para uma página de perfil, o que pode ou não ser o caso. Como você ainda não definiu esta página, você terá outro erro!

Abra as configurações do projeto (**/locallibrary/locallibrary/configurações.py**) e adicione o texto abaixo à parte inferior. Agora, quando você fizer login, você deve ser redirecionado para a página inicial do site por padrão.

# Redirect to home URL after login (Default redirects to /accounts/profile/)

LOGIN\_REDIRECT\_URL = '/'

Copiar para área de transferência

### [Modelo de logout](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#logout_template)

Se você navegar até a URL do logout (http://127.0.0.1:8000/accounts/logout/) então você verá algum comportamento estranho - seu usuário estará logado com certeza, mas você será levado para a página **do logotipo do Admin**. Não é isso que você quer, mesmo que apenas porque o link de login nessa página o leva à tela de login do Administrador (e isso só está disponível para usuários que têm a is\_staff).

Crie e abra **/locallibrary/templates/registration/logged\_out.html**. Copie o texto abaixo:

{% extends "base\_generic.html" %}

{% block content %}

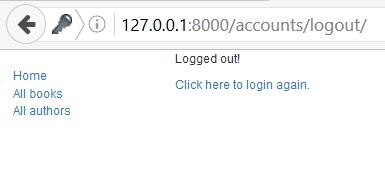
<p>Logged out!</p>

<a href="{% url 'login'%}">Click here to login again.</a>

{% endblock %}

Copiar para área de transferência

Este modelo é muito simples. Ele apenas exibe uma mensagem informando que você foi logado e fornece um link que você pode pressionar para voltar para a tela de login. Se você for para a URL do logout novamente, você deve ver esta página:



### [Modelos de redefinição de senha](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#password_reset_templates)

O sistema padrão de redefinição de senha usa e-mail para enviar ao usuário um link de redefinição. Você precisa criar formulários para obter o endereço de e-mail do usuário, enviar o e-mail, permitir que eles digitem uma nova senha e observar quando todo o processo estiver concluído.

Os modelos a seguir podem ser usados como ponto de partida.

#### Formulário de redefinição de senha

Este é o formulário usado para obter o endereço de e-mail do usuário (para enviar o e-mail de redefinição de senha). Crie **/locallibrary/templates/registration/password\_reset\_form.html** e dê o seguinte conteúdo:

{% extends "base\_generic.html" %}

{% block content %}

<form action="" method="post">

{% csrf\_token %}

{% if form.email.errors %}

{{ form.email.errors }}

{% endif %}

<p>{{ form.email }}</p>

<input type="submit" class="btn btn-default btn-lg" value="Reset password">

</form>

{% endblock %}

Copiar para área de transferência

#### Redefinição de senha feita

Este formulário é exibido após a coleta do seu endereço de e-mail. Crie **/locallibrary/templates/registration/password\_reset\_done.html** e dê o seguinte conteúdo:

{% extends "base\_generic.html" %}

{% block content %}

<p>We've emailed you instructions for setting your password. If they haven't arrived in a few minutes, check your spam folder.</p>

{% endblock %}

Copiar para área de transferência

#### E-mail de redefinição de senha

Este modelo fornece o texto do e-mail HTML contendo o link de reset que enviaremos aos usuários. Crie **/locallibrary/templates/registration/password\_reset\_email.html** e dê o seguinte conteúdo:

Someone asked for password reset for email {{ email }}. Follow the link below:

{{ protocol}}://{{ domain }}{% url 'password\_reset\_confirm' uidb64=uid token=token %}

Copiar para área de transferência

#### Confirmação de redefinição de senha

Esta página é onde você digita sua nova senha depois de clicar no link no e-mail de redefinição de senha. Crie **/locallibrary/templates/registration/password\_reset\_confirm.html** e dê o seguinte conteúdo:

{% extends "base\_generic.html" %}

{% block content %}

{% if validlink %}

<p>Please enter (and confirm) your new password.</p>

<form action="" method="post">

{% csrf\_token %}

<table>

<tr>

<td>{{ form.new\_password1.errors }}

<label for="id\_new\_password1">New password:</label></td>

<td>{{ form.new\_password1 }}</td>

</tr>

<tr>

<td>{{ form.new\_password2.errors }}

<label for="id\_new\_password2">Confirm password:</label></td>

<td>{{ form.new\_password2 }}</td>

</tr>

<tr>

<td></td>

<td><input type="submit" value="Change my password" /></td>

</tr>

</table>

</form>

{% else %}

<h1>Password reset failed</h1>

<p>The password reset link was invalid, possibly because it has already been used. Please request a new password reset.</p>

{% endif %}

{% endblock %}

Copiar para área de transferência

#### Redefinição de senha completa

Este é o último modelo de redefinição de senha, que é exibido para notificá-lo quando a redefinição de senha for bem sucedida. Crie **/locallibrary/templates/registration/password\_reset\_complete.html** e dê o seguinte conteúdo:

{% extends "base\_generic.html" %}

{% block content %}

<h1>The password has been changed!</h1>

<p><a href="{% url 'login' %}">log in again?</a></p>

{% endblock %}

Copiar para área de transferência

### [Testando as novas páginas de autenticação](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#testing_the_new_authentication_pages)

Agora que você adicionou a configuração de URL e criou todos esses modelos, as páginas de autenticação agora devem funcionar apenas!

Você pode testar as novas páginas de autenticação tentando fazer login e, em seguida, sair da sua conta de superusuário usando esses URLs:

* http://127.0.0.1:8000/accounts/login/
* http://127.0.0.1:8000/accounts/logout/

Você poderá testar a funcionalidade de redefinição de senha a partir do link na página de login. **Esteja ciente de que o Django só enviará e-mails de reset para endereços (usuários) que já estão armazenados em seu banco de dados!**

**Nota:** O sistema de redefinição de senha exige que seu site suporte e-mail, que está além do escopo deste artigo, para que esta parte **não funcione ainda**. Para permitir o teste, coloque a seguinte linha no final do seu arquivo settings.py. Isso registra todos os e-mails enviados para o console (para que você possa copiar o link de redefinição de senha do console).

EMAIL\_BACKEND = 'django.core.mail.backends.console.EmailBackend'

Copiar para área de transferência

Para obter mais informações, consulte [Enviar e-mail](https://docs.djangoproject.com/en/4.0/topics/email/) (docs Django).

## [Testes contra usuários autenticados](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#testing_against_authenticated_users)

Esta seção analisa o que podemos fazer para controlar seletivamente o conteúdo que o usuário vê com base em se eles estão conectados ou não.

### [Testes em modelos](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#testing_in_templates)

Você pode obter informações sobre o usuário atualmente conectado em modelos com a variável modelo {{ user }} (isso é adicionado ao contexto do modelo por padrão quando você configura o projeto como fizemos em nosso esqueleto).

Normalmente, você primeiro testará contra a variável {{ user.is\_authenticated }} para determinar se o usuário é elegível para ver conteúdo específico. Para demonstrar isso, atualizaremos nossa barra lateral para exibir um link "Login" se o usuário estiver logado e um link "Logout" se estiver logado.

Abra o modelo base (**/locallibrary/catalog/templates/base\_generic.html**) e copie o texto a seguir no bloco sidebar, imediatamente antes da tag de modelo endblock.

<ul class="sidebar-nav">

...

{% if user.is\_authenticated %}

<li>User: {{ user.get\_username }}</li>

<li><a href="{% url 'logout'%}?next={{request.path}}">Logout</a></li>

{% else %}

<li><a href="{% url 'login'%}?next={{request.path}}">Login</a></li>

{% endif %}

</ul>

Copiar para área de transferência

Como você pode ver, usamos if de modelo else para exibir texto condicionalmente com base em se {{ user.is\_authenticated }} é verdade.endif Se o usuário for autenticado, então sabemos que temos um usuário válido, então {{ user.get\_username }} para exibir seu nome.

Criamos as URLs de link de login e logout usando a tag de modelo url e os nomes das respectivas configurações de URL. Observe também como anexamos ?next={{request.path}} até o final dos URLs. O que isso faz é adicionar um parâmetro de URL next contendo o endereço (URL) da página atual, até o final da URL vinculada. Depois que o usuário tiver logado com sucesso, as visualizações usarão esse valor "next" para redirecionar o usuário de volta para a página onde ele primeiro clicou no link de login/logout.

**Nota:** Experimente! Se você estiver na página inicial e clicar em Login/Logout na barra lateral, então depois que a operação for concluída, você deve voltar à mesma página.

### [Testes em visualizações](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#testing_in_views)

Se você estiver usando visualizações baseadas em funções, a maneira mais fácil de restringir o acesso às suas funções é aplicar o login\_required decorador à sua função de exibição, como mostrado abaixo. Se o usuário estiver logado, seu código de exibição será executado normalmente. Se o usuário não estiver logado, isso redirecionará para a URL de login definida nas configurações do projeto (settings.LOGIN\_URL), passando o caminho absoluto atual como o next parâmetro de URL. Se o usuário conseguir fazer login, então ele será devolvido de volta a esta página, mas desta vez autenticado.

from django.contrib.auth.decorators import login\_required

@login\_required

def my\_view(request):

...

Copiar para área de transferência

**Nota:** Você pode fazer o mesmo tipo de coisa manualmente testando em request.user.is\_authenticated, mas o decorador é muito mais conveniente!

Da mesma forma, a maneira mais fácil de restringir o acesso a usuários conectados em suas visualizações baseadas em classe é derivar do LoginRequiredMixin. Você precisa declarar essa mistura primeiro na lista de superclasse, antes da classe de exibição principal.

from django.contrib.auth.mixins import LoginRequiredMixin

class MyView(LoginRequiredMixin, View):

...

Copiar para área de transferência

Isso tem exatamente o mesmo comportamento de redirecionamento login\_required. Você também pode especificar um local alternativo para redirecionar o usuário para se ele não for autenticado (login\_url) e um nome de parâmetro de URL em vez de "next" para inserir o caminho absoluto atual (redirect\_field\_name).

class MyView(LoginRequiredMixin, View):

login\_url = '/login/'

redirect\_field\_name = 'redirect\_to'

Copiar para área de transferência

Para mais detalhes, confira os [documentos django aqui](https://docs.djangoproject.com/en/4.0/topics/auth/default/#limiting-access-to-logged-in-users).

## [Exemplo — listando os livros do usuário atual](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#example_%E2%80%94_listing_the_current_users_books)

Agora que sabemos como restringir uma página a um determinado usuário, vamos criar uma visão dos livros que o usuário atual pegou emprestado.

Infelizmente, ainda não temos como os usuários emprestarem livros! Então, antes de criarmos a lista de livros, primeiro estenderemos o modelo BookInstance para apoiar o conceito de empréstimo e usar o aplicativo Django Admin para emprestar uma série de livros ao nosso usuário de teste.

### [Modelos](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#models)

Primeiro, teremos que tornar possível que os usuários tenham um BookInstance em empréstimo (já temos um status e uma data due\_back, mas ainda não temos nenhuma associação entre esse modelo e um Usuário. Vamos criar um usando um campo ForeignKey (um para muitos). Também precisamos de um mecanismo fácil para testar se um livro emprestado está atrasado.

Abra **o catálogo/modelos.py** e importe o modelo de User a partir de django.contrib.auth.models (adicione isso logo abaixo da linha de importação anterior na parte superior do arquivo, de modo que o User esteja disponível para código subsequente que faz uso dele):

from django.contrib.auth.models import User

Copiar para área de transferência

Em seguida, adicione o campo borrower ao modelo BookInstance:

borrower = models.ForeignKey(User, on\_delete=models.SET\_NULL, null=True, blank=True)

Copiar para área de transferência

Enquanto estamos aqui, vamos adicionar uma propriedade que podemos chamar de nossos modelos para dizer se uma instância de livro em particular está atrasada. Embora possamos calcular isso no próprio modelo, usar uma [propriedade](https://docs.python.org/3/library/functions.html#property) como mostrado abaixo será muito mais eficiente.

Adicione isso em algum lugar perto do topo do arquivo:

from datetime import date

Copiar para área de transferência

Agora adicione a seguinte definição de propriedade à classe BookInstance:

@property

def is\_overdue(self):

if self.due\_back and date.today() > self.due\_back:

return True

return False

Copiar para área de transferência

**Nota:** Primeiro verificamos se due\_back está vazia antes de fazer uma comparação. Um campo due\_back faria com que Django lançasse um erro em vez de mostrar a página: valores vazios não são comparáveis. Isso não é algo que gostaríamos que nossos usuários experimentassem!

Agora que atualizamos nossos modelos, precisaremos fazer novas migrações no projeto e, em seguida, aplicar essas migrações:

python3 manage.py makemigrations

python3 manage.py migrate

Copiar para área de transferência

### [Admin](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#admin)

Agora abra **catálogo/administrador.py** e adicione o campo borrower à classe BookInstanceAdmin tanto na list\_display quanto nos fieldsets, conforme mostrado abaixo. Isso tornará o campo visível na seção Administrador, permitindo-nos atribuir um User a um BookInstance quando necessário.

@admin.register(BookInstance)

class BookInstanceAdmin(admin.ModelAdmin):

list\_display = ('book', 'status', 'borrower', 'due\_back', 'id')

list\_filter = ('status', 'due\_back')

fieldsets = (

(None, {

'fields': ('book','imprint', 'id')

}),

('Availability', {

'fields': ('status', 'due\_back','borrower')

}),

)

Copiar para área de transferência

### [Empresto alguns livros](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#loan_a_few_books)

Agora que é possível emprestar livros para um usuário específico, vá e emprese uma série de registros BookInstance. Defina seu campo borrowed ao seu usuário de teste, faça o status "No empréstimo", e defina datas de vencimento tanto no futuro quanto no passado.

**Nota:** Nós não vamos soletrar o processo, como você já sabe como usar o site do Administrador!

### [Na vista do empréstimo](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#on_loan_view)

Agora vamos adicionar uma visão para obter a lista de todos os livros que foram emprestados para o usuário atual. Usaremos a mesma exibição de lista genérica baseada em classe que estamos familiarizados, mas desta vez também importaremos e derivaremos do LoginRequiredMixin, de modo que apenas um usuário conectado pode chamar essa exibição. Também optaremos por declarar um template\_name, em vez de usar o padrão, pois podemos acabar tendo algumas listas diferentes de registros bookInstance, com diferentes visualizações e modelos.

Adicione o seguinte ao catálogo/visualizações.py:

from django.contrib.auth.mixins import LoginRequiredMixin

class LoanedBooksByUserListView(LoginRequiredMixin,generic.ListView):

"""Generic class-based view listing books on loan to current user."""

model = BookInstance

template\_name ='catalog/bookinstance\_list\_borrowed\_user.html'

paginate\_by = 10

def get\_queryset(self):

return BookInstance.objects.filter(borrower=self.request.user).filter(status\_\_exact='o').order\_by('due\_back')

Copiar para área de transferência

A fim de restringir nossa consulta apenas aos objetos BookInstance para o usuário atual, reassuramos get\_queryset() como mostrado acima. Observe que "o" é o código armazenado para "emprestado" e pedimos até a data due\_back para que os itens mais antigos sejam exibidos primeiro.

### [URL conf para em livros de empréstimos](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#url_conf_for_on_loan_books)

Agora abra **/catalog/urls.py** e adicione um path() apontando para a exibição acima (você pode apenas copiar o texto abaixo até o final do arquivo).

urlpatterns += [

path('mybooks/', views.LoanedBooksByUserListView.as\_view(), name='my-borrowed'),

]

Copiar para área de transferência

### [Modelo para livros de empréstimo](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#template_for_on-loan_books)

Agora, tudo o que precisamos fazer para esta página é adicionar um modelo. Primeiro, crie o arquivo de modelo **/catalog/templates/catalog/bookinstance\_list\_borrowed\_user.html** e dê-lhe o seguinte conteúdo:

{% extends "base\_generic.html" %}

{% block content %}

<h1>Borrowed books</h1>

{% if bookinstance\_list %}

<ul>

{% for bookinst in bookinstance\_list %}

<li class="{% if bookinst.is\_overdue %}text-danger{% endif %}">

<a href="{% url 'book-detail' bookinst.book.pk %}">{{bookinst.book.title}}</a> ({{ bookinst.due\_back }})

</li>

{% endfor %}

</ul>

{% else %}

<p>There are no books borrowed.</p>

{% endif %}

{% endblock %}

Copiar para área de transferência

Este modelo é muito semelhante ao que criamos anteriormente para os objetos Book e Author. A única coisa "nova" aqui é que verificamos o método que adicionamos no modelo (bookinst.is\_overdue) e usamos para mudar a cor dos itens vencidos.

Quando o servidor de desenvolvimento estiver em execução, agora você deve ser capaz de exibir a lista de um usuário logado no seu navegador http://127.0.0.1:8000/catalog/mybooks/. Experimente isso com seu usuário logado e logado (no segundo caso, você deve ser redirecionado para a página de login).

### [Adicione a lista à barra lateral](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#add_the_list_to_the_sidebar)

O último passo é adicionar um link para esta nova página na barra lateral. Colocaremos isso na mesma seção onde exibimos outras informações para o usuário logado.

Abra o modelo base (**/locallibrary/catalog/templates/base\_generic.html**) e adicione a linha "Meu Emprestado" à barra lateral na posição mostrada abaixo.

<ul class="sidebar-nav">

{% if user.is\_authenticated %}

<li>User: {{ user.get\_username }}</li>

<li><a href="{% url 'my-borrowed' %}">My Borrowed</a></li>

<li><a href="{% url 'logout'%}?next={{request.path}}">Logout</a></li>

{% else %}

<li><a href="{% url 'login'%}?next={{request.path}}">Login</a></li>

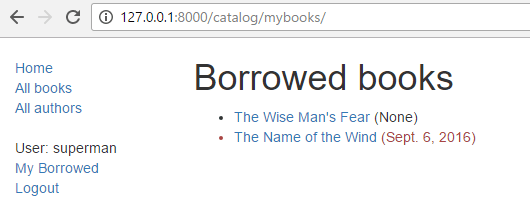
{% endif %}

</ul>

Copiar para área de transferência

### [O que parece?](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#what_does_it_look_like)

Quando qualquer usuário estiver logado, ele verá o link Meu Emprestado na barra lateral, e a lista de livros exibidos como abaixo (o primeiro livro não tem data de vencimento, que é um bug que esperamos corrigir em um tutorial posterior!).



## [Permissões](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#permissions)

As permissões estão associadas aos modelos e definem as operações que podem ser realizadas em uma instância de modelo por um usuário que tem a permissão. Por padrão, o Django automaticamente dá permissões adicionais, alterações e exclusão para todos os modelos, que permitem aos usuários com as permissões realizar as ações associadas através do site administrativo. Você pode definir suas próprias permissões para modelos e concedê-las a usuários específicos. Você também pode alterar as permissões associadas a diferentes instâncias do mesmo modelo.

Testar permissões em visualizações e modelos é então muito semelhante a testes no status de autenticação (e, de fato, testar uma permissão também testa para autenticação).

### [Modelos](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#models_2)

A definição de permissões é feita na seção modelo "class Meta", usando o campo permissions. Você pode especificar quantas permissões precisar em uma tupla, cada permissão sendo definida em uma tupla aninhada contendo o nome de permissão e o valor de exibição de permissão. Por exemplo, podemos definir uma permissão para permitir que um usuário marque que um livro foi devolvido como mostrado:

class BookInstance(models.Model):

...

class Meta:

...

permissions = (("can\_mark\_returned", "Set book as returned"),)

Copiar para área de transferência

Poderíamos então atribuir a permissão a um grupo "Bibliotecário" no site do Administrador.

Abra o **catálogo/modelos.py** e adicione a permissão como mostrado acima. Você precisará re-executar suas migrações (ligue para python3 manage.py makemigrations e python3 manage.py migrate) para atualizar o banco de dados adequadamente.

### [Modelos](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#templates)

As permissões do usuário atual são armazenadas em uma variável de modelo chamada {{ perms }}. Você pode verificar se o usuário atual tem uma permissão específica usando o nome de variável específico dentro do "app" Django associado — por exemplo{{ perms.catalog.can\_mark\_returned }} será True se o usuário tiver essa permissão, e False de outra forma. Normalmente testamos a permissão usando o modelo {% if %} tag como mostrado:

{% if perms.catalog.can\_mark\_returned %}

<!-- We can mark a BookInstance as returned. -->

<!-- Perhaps add code to link to a "book return" view here. -->

{% endif %}

Copiar para área de transferência

### [Modos de exibição](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#views)

As permissões podem ser testadas na visualização de função usando o permission\_required ou em uma exibição baseada em classe usando a PermissionRequiredMixin. O padrão é o mesmo da autenticação de login, embora, é claro, você possa razoavelmente ter que adicionar várias permissões.

Decorador de visualização de funções:

from django.contrib.auth.decorators import permission\_required

@permission\_required('catalog.can\_mark\_returned')

@permission\_required('catalog.can\_edit')

def my\_view(request):

...

Copiar para área de transferência

Uma mistura necessária para visualizações baseadas em classe.

from django.contrib.auth.mixins import PermissionRequiredMixin

class MyView(PermissionRequiredMixin, View):

permission\_required = 'catalog.can\_mark\_returned'

# Or multiple permissions

permission\_required = ('catalog.can\_mark\_returned', 'catalog.can\_edit')

# Note that 'catalog.can\_edit' is just an example

# the catalog application doesn't have such permission!

Copiar para área de transferência

**Nota:** Há uma pequena diferença padrão no comportamento acima. Por **padrão** para um usuário de login com uma violação de permissão:

* @permission\_required redireciona para a tela de login (STATUS HTTP 302).
* PermissionRequiredMixin retorna 403 (STATUS HTTP Proibido).

Normalmente você vai querer o comportamento PermissionRequiredMixin: retornar 403 se um usuário estiver logado, mas não tiver a permissão correta. Para fazer isso para uma exibição de função use @login\_required e @permission\_required com raise\_exception=True como mostrado:

from django.contrib.auth.decorators import login\_required, permission\_required

@login\_required

@permission\_required('catalog.can\_mark\_returned', raise\_exception=True)

def my\_view(request):

...

Copiar para área de transferência

### [Exemplo](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#example)

Não atualizaremos o LocalLibrary aqui; talvez no próximo tutorial!

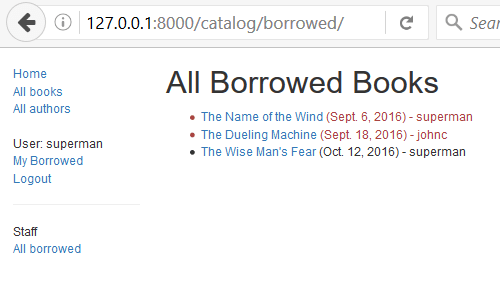
## [Desafie-se](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#challenge_yourself)

No início deste artigo, mostramos como criar uma página para o usuário atual, listando os livros que eles pegaram emprestado. O desafio agora é criar uma página semelhante que só seja visível para bibliotecários, que exibe todos os livros que foram emprestados, e que inclui o nome de cada mutuário.

Você deve ser capaz de seguir o mesmo padrão que para a outra visão. A principal diferença é que você precisará restringir a visão apenas para bibliotecários. Você pode fazer isso com base em se o usuário é um membro da equipe (decorador de funções: staff\_member\_required, variável de modelo: user.is\_staff), mas recomendamos que você use a permissão can\_mark\_returned e PermissionRequiredMixin, conforme descrito na seção anterior.

**Aviso:** Lembre-se de não usar seu superusuário para testes baseados em permissões (verificações de permissão sempre retornam verdadeiros para superusuários, mesmo que uma permissão ainda não tenha sido definida!). Em vez disso, crie um usuário bibliotecário e adicione o recurso necessário.

Quando você terminar, sua página deve se parecer com a captura de tela abaixo.



## [Resumo](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#summary)

Excelente trabalho — agora você criou um site onde os membros da biblioteca podem fazer login e visualizar seu próprio conteúdo, e onde os bibliotecários (com a permissão correta) podem visualizar todos os livros emprestados e seus mutuários. No momento ainda estamos apenas visualizando conteúdo, mas os mesmos princípios e técnicas são usados quando você quer começar a modificar e adicionar dados.

Em nosso próximo artigo, veremos como você pode usar formulários Django para coletar informações do usuário e, em seguida, começar a modificar alguns de nossos dados armazenados.

## [Veja também](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#see_also)

* [Autenticação do usuário em Django](https://docs.djangoproject.com/en/4.0/topics/auth/) (docs Django)
* [Usando o (padrão) sistema de autenticação Django](https://docs.djangoproject.com/en/4.0/topics/auth/default/) (docs Django)
* [Introdução a visualizações baseadas em classe > decorar visualizações baseadas em classe](https://docs.djangoproject.com/en/4.0/topics/class-based-views/intro/#decorating-class-based-views) (docs Django)
* [Anterior](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Sessions)
* [Visão geral: Django](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django)
* [Próximo](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Forms)

## [Neste módulo](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication#in_this_module)

* [Introdução django](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction)
* [Criação de um ambiente de desenvolvimento Django](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/development_environment)
* [Tutorial django: o site da Biblioteca Local](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial_local_library_website)
* [Django Tutorial Parte 2: Criando um site de esqueletos](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/skeleton_website)
* [Tutorial Django Parte 3: Usando modelos](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Models)
* [Django Tutorial Parte 4: Site de administração de Django](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Admin_site)
* [Tutorial Django Parte 5: Criando nossa página inicial](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Home_page)
* [Django Tutorial Parte 6: Lista genérica e visualizações detalhadas](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Generic_views)
* [Django Tutorial Parte 7: Quadro de sessões](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Sessions)
* **Django Tutorial Parte 8: Autenticação e permissões do usuário**
* [Django Tutorial Parte 9: Trabalhando com formulários](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Forms)
* [Django Tutorial Parte 10: Testando um aplicativo web Django](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Testing)
* [Django Tutorial Parte 11: Implantação de Django para a produção](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Deployment)
* [Segurança de aplicativos web django](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/web_application_security)
* [DIY Django mini blog](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/django_assessment_blog)

### Encontrou um problema com esta página?

* [Editar no **GitHub**](https://github.com/mdn/content/edit/main/files/en-us/learn/server-side/django/authentication/index.md)
* [Fonte no **GitHub**](https://github.com/mdn/content/blob/main/files/en-us/learn/server-side/django/authentication/index.md?plain=1)
* [Reporte um problema com este conteúdo no **GitHub**](https://github.com/mdn/content/issues/new?template=page-report.yml&mdn-url=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FLearn%2FServer-side%2FDjango%2FAuthentication&metadata=%3C%21--+Do+not+make+changes+below+this+line+--%3E%0A%3Cdetails%3E%0A%3Csummary%3EPage+report+details%3C%2Fsummary%3E%0A%0A*+Folder%3A+%60en-us%2Flearn%2Fserver-side%2Fdjango%2Fauthentication%60%0A*+MDN+URL%3A+https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FLearn%2FServer-side%2FDjango%2FAuthentication%0A*+GitHub+URL%3A+https%3A%2F%2Fgithub.com%2Fmdn%2Fcontent%2Fblob%2Fmain%2Ffiles%2Fen-us%2Flearn%2Fserver-side%2Fdjango%2Fauthentication%2Findex.md%0A*+Last+commit%3A+https%3A%2F%2Fgithub.com%2Fmdn%2Fcontent%2Fcommit%2F59292882a10919dec693e619b75e2c9d8018f07c%0A*+Document+last+modified%3A+2022-04-07T10%3A22%3A34.000Z%0A%0A%3C%2Fdetails%3E)
* Quer resolver o problema você mesmo? Veja [nosso guia de Contribuição](https://github.com/mdn/content/blob/main/README.md).

**Última modificação:** Apr 7, 2022, [por contribuintes do MDN](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication/contributors.txt)