

# django

## Tutorial de criação de aplicação Django com PostgreSQL

Cleuton Sampaio: <http://pythondrops.com>

## Índice

O que é o Django?	3
Estudo de caso	3
Preparar o ambiente do projeto	4
Ambiente virtual	4
Banco de dados	5
Criando a aplicação Django	6
Configurando o banco de dados	7
Modelo	8
Criando o modelo	10
O site administrativo	14
Usando o Shell	16
Interface de usuário	18
Autenticação e controle de acesso	22
Uso de sessão	24
Terminando a aplicação	26
Conclusão	32
Devo utilizar Django?	33

## O que é o Django?

É um framework para desenvolvimento de aplicações Web em Python, que abstrai grande parte do trabalho básico (boilerplate) permitindo que você foque na funcionalidade da sua aplicação.

Como todo framework, ele dirige a maneira como você programa, restringindo sua liberdade de implementação e as suas escolhas. Antes de adotar um framework como o Django, você deve avaliar o impacto dessa restrição no roadmap da sua aplicação.

A melhor maneira de aprender uma tecnologia é implementando algo com ela, e faremos assim com o Django. Melhor do que ficarmos explicando conceitos abstratos com exemplos vagos.

O repositório dessa aplicação é: <https://github.com/cleuton/djangosample>

## Estudo de caso

Vamos criar uma aplicação de quadro de recados. Sabe aqueles quadros de cortiça, que ficam nas paredes das escolas e lojas? Onde você pode pregar seu anúncio ou recado? Então, vamos criar uma versão online disso.

Você se registra e faz login e tem acesso ao seu quadro pessoal de recados, onde aparecerão os recados que você postou e aqueles que as pessoas que você segue postaram. A princípio, você pode postar quantos recados quiser e também pode apagar os recados que postou. Para não complicar demais, vamos apenas permitir apagar os recados próprios e não editá-los.

Um recado é uma registro como esse:

- Id: Identificador do recado, sua chave primária;
- Título: Título do recado;
- Data: Data de postagem;
- Texto: Texto do recado;
- Autor: Usuário autor do recado (chave estrangeira);

Você pode seguir autores, neste caso, precisamos guardar essa informação:

- Id\_Usuario: Sua identificação;
- Id\_Alvo: O usuário que você segue;

Ao logar na aplicação, aparecem as informações:

1. Quantos usuários seguem você;
2. Quantos usuários você segue;
3. Seus recados, em ordem decrescente cronológica. Os recados que você postou permitem deleção. Só título, data e nome do autor;
4. Ao clicar em um título, você lê o resto do recado postado.

Na tela principal há um link para postar um novo recado.

Simples, não?

## Preparar o ambiente do projeto

Antes de começar a trabalhar em um projeto Python, devemos preparar o ambiente. Você vai necessitar do Python 3.x. Eu recomendo no mínimo o 3.8, e vai precisar criar um ambiente virtual para não “estragar” seu ambiente Python principal.

Ah, e vamos utilizar o PostgreSQL, mas não se preocupe, pois não vamos instalar nada na sua máquina. Para isto, vamos utilizar o Docker! Instale o Docker conforme esse site:

<https://docs.docker.com/get-started/>

A instalação é muito simples e ele funciona em Microsoft Windows, Apple MacOS ou Linux. Todo desenvolvedor tem que ter um Docker instalado.

A outra opção é você instalar um PostgreSQL ou utilizar o SQLite...

Neste exemplo, vou utilizar o PostgreSQL, então você pode instalá-lo com as instruções deste site:

<https://www.postgresql.org/download/>

Mas acredite: Instalar o Docker vale mais a pena!

## Ambiente virtual

Escolha uma pasta para criar todo o projeto. De preferência, que seja recém criada (deve estar vazia). Abra o Terminal ou Prompt de comandos, entre na pasta e digite:

```
python -m venv .
```

Isto criará um ambiente virtual naquela pasta. Ele pode ser ativado com o comando:

### **Linux:**

```
source bin/activate
```

### **MS Windows (cmd):**

```
\Scripts\activate.bat
```

### **MS Windows (Powershell):**

```
\Scripts\Activate.ps1
```

O ambiente virtual pode ser desativado fechando a janela ou então rodando o comando: deactivate.

Tudo o que você instalar no ambiente virtual ativado só existirá nesse ambiente virtual. Use o PIP normalmente.

Eu recomendo criar um arquivo “requirements.txt” na pasta do projeto com esse conteúdo:

```
Django
gunicorn
psycopg2-binary
```

O gunicorn é para servir a aplicação em “produção”. Durante o desenvolvimento vamos utilizar o servidor web do Django mesmo.

## Banco de dados

Agora, vamos subir um PostgreSQL com o Docker:

```
sudo docker run --name some-postgres -p 5432:5432 -e POSTGRES_PASSWORD=password
-d postgres
```

Não pule linha ao digitar isso no terminal.

- *É possível eliminar a necessidade do uso de “sudo” em comandos “docker”. Basta seguir essa recomendação: <https://github.com/sindresorhus/guides/blob/main/docker-without-sudo.md>*

Isto subirá um PostgreSQL em Localhost, que escutará na porta 5432.

Agora, vamos abrir um SSH para o contêiner e usar o “psql” para criar um banco de dados nele. Primeiramente, vamos usar o docker para iniciar uma sessão terminal no contêiner:

```
docker exec -it some-postgres /bin/bash
```

Para sair da sessão, use o comando bash “exit”.

Agora, vamos rodar o “psql” com o usuário “postgres”:

```
psql -U postgres
```

Isso abrirá um shell SQL para o servidor PostgreSQL. Então, podemos rodar um comando para criar um banco de dados:

```
create database recadosdb;
```

Note o ponto e vírgula no final do comando! Comandos SQL precisam de ponto e vírgula, mas outros comandos não. Por exemplo, uma vez criado o banco de dados, podemos sair do “psql” com o comando: “\q” sem ponto e vírgula. Depois, usamos o “exit” e pronto!

## Criando a aplicação Django

Bom, sabe como são os frameworks, não? Se tentar entender tudo, vai entender nada! Simplesmente siga a correnteza e, quando menos desconfiar, já estará sabendo usar o Django.

Primeiramente, ative o ambiente virtual e rode o PIP para instalar as dependências:

```
pip install -r requirements.txt
```

O Django só estará instalado no seu ambiente virtual, ou seja, para rodar a aplicação tem que ativar o ambiente virtual.

Vamos criar um projeto Django, que pode conter uma ou mais aplicações Django:

```
django-admin startproject recados
```

Mas o que seria isso? Ele criou uma pasta (sob a pasta que você está, aquela que tem o “requirements.txt”) com esse conteúdo:

```
.
./recados
    manage.py
    recados
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
```

Há uma pasta “recados”, que contém uma subpasta “recados”...

A pasta “recados” superior é a pasta do nosso projeto. Na verdade, você pode até renomeá-la, se quiser. Note que ela tem o script “**manage.py**”, com o qual você fará quase tudo no seu projeto!

A subpasta “recados” é o pacote Python do seu projeto. Ela contém o “\_\_init\_\_.py” o que a torna um pacote, e vários scripts:

- asgi.py: Se você decidir servir sua aplicação utilizando ASGI (Python assíncrono) este é o entry-point;
- settings.py: A configuração do projeto Django. Você fará muita coisa nesse script;
- urls.py: A declaração das rotas (ou URLs) do seu projeto;
- wsgi.py: Se você decidir servir sua aplicação utilizando WSGI (Gunicorn, por exemplo) este é o entry-point;

Você criou o projeto, agora vamos criar a aplicação. Dentro do projeto “recados”, vamos criar uma aplicação “quadro”, que servirá os recados. Em um projeto, podemos ter várias aplicações diferentes, que compartilham uma mesma configuração básica.

A pasta do projeto é a pasta onde está o script “manage.py”. Vá para ela e use o próprio “manage.py” para criar uma aplicação. Há momentos em que você vai usar o “django-admin”, mas é mais fácil usar o “manage.py” para certas tarefas internas do projeto. Digite:

```
python manage.py startapp quadro
```

Agora, na pasta “recados” temos esta estrutura:

```
./recados:
  quadro:
    migrations:
      __init__.py
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  recados:
  ...
```

- **migrations:** As atualizações que precisam ser feitas no banco de dados devido às alterações que você fez no modelo da aplicação;
- **admin.py:** Utilizado para exibir seu modelo no painel administrativo do Django. É um CRUD básico;
- **apps.py:** Para você criar configurações da sua aplicação;
- **models.py:** Para você criar o modelo da sua aplicação. As classes que representam entidades. Alterações aqui precisam ser sincronizadas com o banco de dados através de migrações;
- **tests.py:** Aqui você cria os testes da aplicação;
- **views.py:** Aqui você cria as views ou as respostas da aplicação às URLs.

## Configurando o banco de dados

Por padrão, o Django utiliza o SQLite. Eu não quero isso, pois preciso criar um site que usa PostgreSQL.

Então, temos que configurar o banco de dados que vamos utilizar. Nós já subimos o PostgreSQL e criamos o banco de dados, agora precisamos configurar isso no Django. Onde? No arquivo de configurações do projeto, que fica na pasta do projeto, na subpasta com o nome do projeto. No nosso caso é: “recados/recados/settings.py”. Localize a variável “DATABASES”:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
```

```
}  
}
```

Como eu não quero usar o SQLite, tenho que configurar tudo aqui. Mude para isto:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'HOST': 'localhost',  
        'NAME': 'recadosdb',  
        'PORT': 5432,  
        'USER': 'postgres',  
        'PASSWORD': 'password'  
    }  
}
```

Lembre-se que você instalou o “Psycopg2” no seu “requirements.txt”! O “host” é o IP ou hostname (ou DNS) do seu servidor PostgreSQL, o “name” é o nome do banco de dados, “port” é a porta do servidor, “user” é o usuário do PostgreSQL e “password” é a senha.

## Modelo

O modelo da sua aplicação representa as entidades com as quais ela lidará. É muito importante começarmos por ele. Você verá que é bastante simples trabalhar com modelos no Django. Depois, vamos fazer a migração para o banco de dados.

Antes de mais nada, o Django já tem um modelo administrativo que será migrado também. Para ver isso, abra a pasta do projeto, e digite o comando (lembre-se de iniciar o PostgreSQL no Docker):

```
python manage.py runserver
```

Isso executará o servidor de desenvolvimento rodando sua aplicação. Você deve ver uma mensagem assim:

```
You have 18 unapplied migration(s). Your project may not work properly until you  
apply the migrations for app(s): admin, auth, contenttypes, sessions.  
Run 'python manage.py migrate' to apply them.
```

Como assim? Eu tenho 18 migrações? Eu não criei nada... Abra a pasta “recados”, dentro da pasta do projeto (que também é “recados”) e veja o arquivo “settings.py”. Procure por “INSTALLED\_APPS” e você verá isso:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',
```



]

Estas aplicações são padrões do Django e estão instaladas em seu projeto, onde também será instalada a sua aplicação. Portanto, ao executar o servidor, ele detectou que estas aplicações também têm modelos e eles não estão refletidos no banco de dados.

As migrações são scripts Python que, se forem relativas ao modelo da sua aplicação, ficarão dentro da pasta “migrations”, dentro da pasta da aplicação Django.

Podemos fazer o que a mensagem de “warning” recomenda e rodar as migrações agora. Isso criará algumas tabelas no nosso banco de dados PostgreSQL. Na pasta do projeto (onde está o “manage.py”) rode o comando:

```
python manage.py migrate
```

O resultado deve ser algo assim:

Running migrations:

```
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying admin.0003_logentry_add_action_flag_choices... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying auth.0012_alter_user_first_name_max_length... OK
Applying sessions.0001_initial... OK
```

E, se examinarmos o banco de dados, veremos essas migrações refletidas nele. Entre no contêiner do PostgreSQL:

```
docker exec -it some-postgres /bin/bash
```

Abrimos o “psql”:

```
psql -U postgres
```

Conectamos ao nosso database:

```
\c recadosdb
```

E listamos as tabelas existentes:

\dt

Você deve ver algo assim:

List of relations			
Schema	Name	Type	Owner
public	auth_group	table	postgres
public	auth_group_permissions	table	postgres
public	auth_permission	table	postgres
public	auth_user	table	postgres
public	auth_user_groups	table	postgres
public	auth_user_user_permissions	table	postgres
public	django_admin_log	table	postgres
public	django_content_type	table	postgres
public	django_migrations	table	postgres
public	django_session	table	postgres
(10 rows)			

Temos várias tabelas criadas. Como eu disse, elas vieram das aplicações que o Django instalou em nosso projeto.

Para sair do “psql” digite: “\q” e, para fechar a sessão com o contêiner, digite: “exit”.

## Criando o modelo

Já temos uma ideia do modelo da aplicação, então podemos criar classes para representar nossas entidades de dados. Vamos rever o modelo aqui:

Um recado é uma registro como esse:

- Id: Identificador do recado, sua chave primária;
- Título: Título do recado;
- Data: Data de postagem;
- Texto: Texto do recado;
- Autor: Usuário autor do recado (chave estrangeira);

Você pode seguir autores, neste caso, precisamos guardar essa informação:

- Id\_Usuario: Sua identificação;
- Id\_Alvo: O usuário que você segue;

Ok... Não está faltando nada? Falamos em “usuário” mas de onde virá isso? Não precisamos criar uma tabela para representar o usuário? Não! O Django já criou para nós:

```
\d+ auth_user
```

Table

```
"public.auth_user"
```

Column Default	Type Storage	Collation Compression	Nullable Stats target	Description
id	integer		not null	
nextval('auth_user_id_seq'::regclass)		plain		
password	character varying(128)		not null	
extended				
last_login	timestamp with time zone			
plain				
is_superuser	boolean		not null	
plain				
username	character varying(150)		not null	
extended				
first_name	character varying(150)		not null	
extended				
last_name	character varying(150)		not null	
extended				
email	character varying(254)		not null	
extended				
is_staff	boolean		not null	
plain				
is_active	boolean		not null	
plain				
date_joined	timestamp with time zone		not null	
plain				

Indexes:

"auth\_user\_pkey" PRIMARY KEY, btree (id)

"auth\_user\_username\_6821ab7c\_like" btree (username varchar\_pattern\_ops)

"auth\_user\_username\_key" UNIQUE CONSTRAINT, btree (username)

Temos uma tabela “auth\_user” que representará o usuário atualmente “logado” na aplicação (sim, haverá “login”). Podemos usar essa tabela para isso.

Começemos com a entidade “recado”. Edite o script “models.py”, que fica dentro da pasta da aplicação “quadro” e acrescente:

```
from django.db import models

class Recado(models.Model):
    titulo = models.CharField(max_length=100, blank=False)
    data = models.DateTimeField(auto_now_add=True)
    texto = models.TextField
    autor = models.ForeignKey('auth.User', related_name='recados',
                              on_delete=models.CASCADE)
```

(o import já deve estar lá)

Pera aí! Cadê o campo “id”? Calma... O Django vai criar automaticamente uma chave primária “id”. Podemos mudar isso, se desejarmos. A classe “models” tem modelos para vários tipos de campo:

- título: Campo texto obrigatório de 100 posições;
- data: Campo de data e hora, que recebe a data e hora de quando o registro foi criado;
- texto: Campo para texto livre sem limite de tamanho;
- autor: Chave estrangeira para a tabela “auth\_user”, que permite deletar este registro quando o usuário for deletado;

Parece simples, não? E é! Agora, vamos à entidade “seguidor”, que indica quais usuários você segue:

```
class Seguidor(models.Model):
    usuario_seguidor = models.ForeignKey('auth.User', related_name='+',
                                         on_delete=models.CASCADE)
    usuario_seguido = models.ForeignKey('auth.User', related_name='+',
                                       on_delete=models.CASCADE)
```

Poderíamos criar uma chave primária composta, mas isso não mudaria muita coisa. Vamos ver como o Django gera isso e depois podemos modificar. Eu coloquei “+” para não deixar o Django gerar o relacionamento reverso.

Os modelos estão criados, mas temos que realizar a migração. Para isso, são necessários alguns passos. Para começar, temos que fazer o projeto ficar ciente da nossa aplicação, e fazemos isso alterando o “settings.py” do projeto:

```
INSTALLED_APPS = [
    'quadro.apps.QuadroConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Nossa aplicação tem um arquivo “apps.py”, onde é criada uma classe “QuadroConfig”:

```
class QuadroConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'quadro'
```

Ao incluir essa classe na variável “INSTALLED\_APPS” estamos informando ao projeto sobre a nossa aplicação.

Para criarmos as migrações necessárias a serem feitas no banco de dados, precisamos rodar um comando:

```
python manage.py makemigrations quadro
```

Isso vai criar alguns scripts python dentro da pasta “migrations” da nossa aplicação. Eis o resultado desse comando:

```
Migrations for 'quadro':
  quadro/migrations/0001_initial.py
    - Create model Seguidor
    - Create model Recado
```

Se editarmos o script “migrations/0001\_initial.py” veremos os comandos para criar o modelo no banco de dados.

Mas podemos fazer algo mais interessante: Ver o SQL que será executado:

```
python manage.py sqlmigrate quadro 0001
```

O resultado é esse:

```
BEGIN;
--
-- Create model Seguidor
--
CREATE TABLE "quadro_seguidor" ("id" bigserial NOT NULL PRIMARY KEY,
"usuario_seguido_id" integer NOT NULL, "usuario_seguidor_id" integer NOT NULL);
--
-- Create model Recado
--
CREATE TABLE "quadro_recado" ("id" bigserial NOT NULL PRIMARY KEY, "titulo"
varchar(100) NOT NULL, "data" timestamp with time zone NOT NULL, "texto" DEFAULT
"" NOT NULL, "autor_id" integer NOT NULL);
ALTER TABLE "quadro_seguidor" ADD CONSTRAINT
"quadro_seguidor_usuario_seguido_id_f8372029_fk_auth_user_id" FOREIGN KEY
("usuario_seguido_id") REFERENCES "auth_user" ("id") DEFERRABLE INITIALLY
DEFERRED;
ALTER TABLE "quadro_seguidor" ADD CONSTRAINT
"quadro_seguidor_usuario_seguidor_id_42ee9ca7_fk_auth_user_id" FOREIGN KEY
("usuario_seguidor_id") REFERENCES "auth_user" ("id") DEFERRABLE INITIALLY
DEFERRED;
CREATE INDEX "quadro_seguidor_usuario_seguido_id_f8372029" ON "quadro_seguidor"
("usuario_seguido_id");
CREATE INDEX "quadro_seguidor_usuario_seguidor_id_42ee9ca7" ON "quadro_seguidor"
("usuario_seguidor_id");
ALTER TABLE "quadro_recado" ADD CONSTRAINT
"quadro_recado_autor_id_ee756b36_fk_auth_user_id" FOREIGN KEY ("autor_id")
REFERENCES "auth_user" ("id") DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "quadro_recado_autor_id_ee756b36" ON "quadro_recado" ("autor_id");
COMMIT;
```

Ele criou as tabelas, colocou os constraints corretos e até criou os índices! Muito legal.

Mas, para isso virar realidade, temos que executar a migração:  
`python manage.py migrate`

A melhor maneira de conferir é rodando o “psql”, como fizemos no início desse tópico. Eis o resultado:

List of relations			
Schema	Name	Type	Owner

---

public	auth_group	table	postgres
public	auth_group_permissions	table	postgres
public	auth_permission	table	postgres
public	auth_user	table	postgres
public	auth_user_groups	table	postgres
public	auth_user_user_permissions	table	postgres
public	django_admin_log	table	postgres
public	django_content_type	table	postgres
public	django_migrations	table	postgres
public	django_session	table	postgres
public	quadro_recado	table	postgres
public	quadro_seguidor	table	postgres
(12 rows)			

Aí estão nossas tabelas.

## O site administrativo

Sua aplicação já tem um site administrativo, como já vimos. Podemos executá-lo e fazer login. Mas, para isso, precisamos de um usuário com poderes administrativos e vamos criar um:

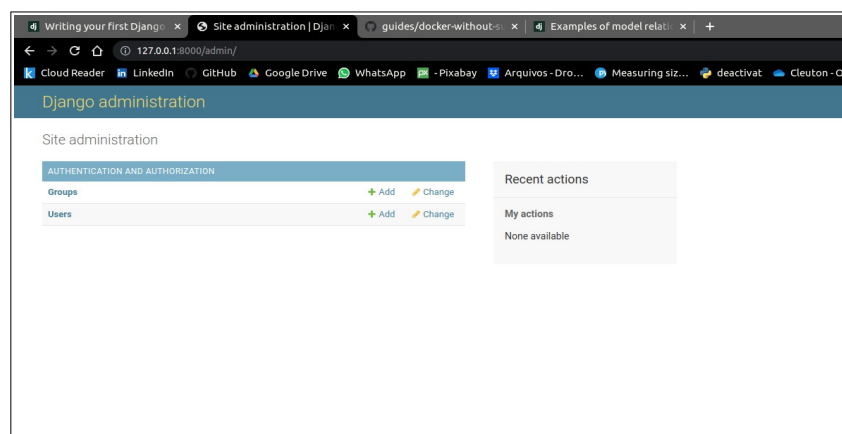
```
python manage.py createsuperuser
```

Entre o username, um email e a senha. Por exemplo: “admin”, “123456” (senha fraca ele vai pedir para você confirmar que deseja utilizá-la).

Agora, rode o servidor de desenvolvimento novamente:

```
python manage.py runserver
```

Abra um navegador e digite a URL: <http://127.0.0.1:8000/admin/> informe o usuário e a senha. Você verá isso:



Nesta página você pode adicionar grupos e usuários da sua aplicação. Mas cadê o nosso modelo?

A aplicação “admin” precisa saber como administrar nosso modelo. Para isso, existe um script “admin.py” dentro da pasta da nossa aplicação, e basta registrarmos nosso modelo:

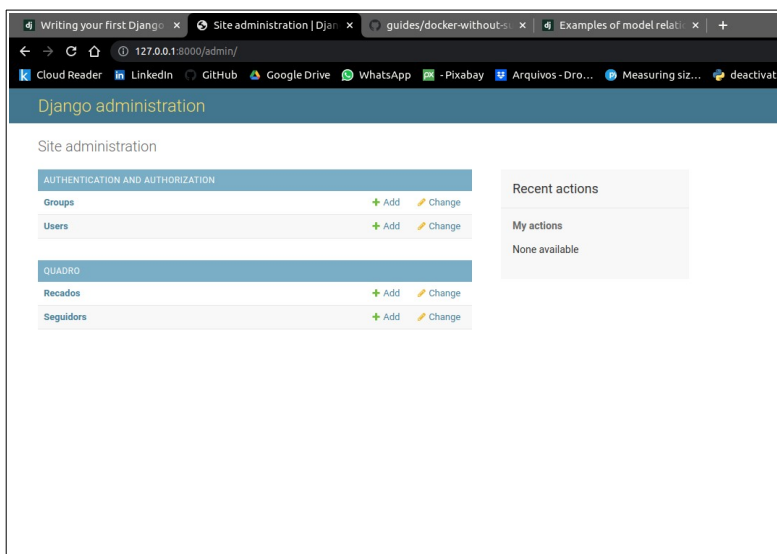
```
from django.contrib import admin

# Register your models here.

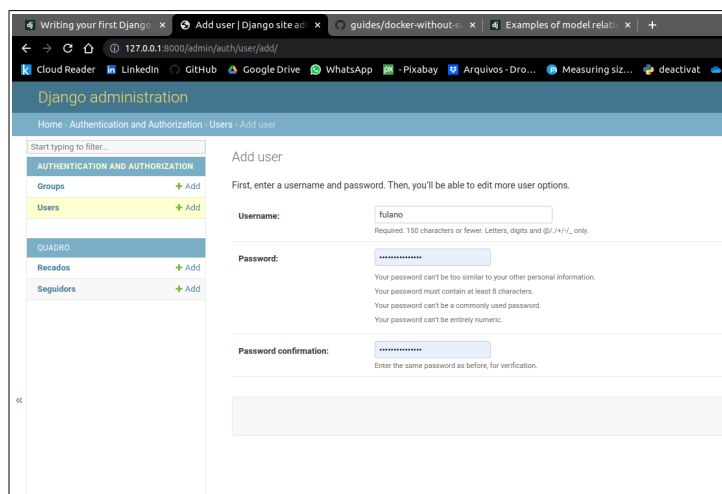
from .models import Recado, Seguidor

admin.site.register(Recado)
admin.site.register(Seguidor)
```

Nem precisa reiniciar o servidor, pois ele pegará as alterações. Basta dar refresh na página de administração e você verá isso:

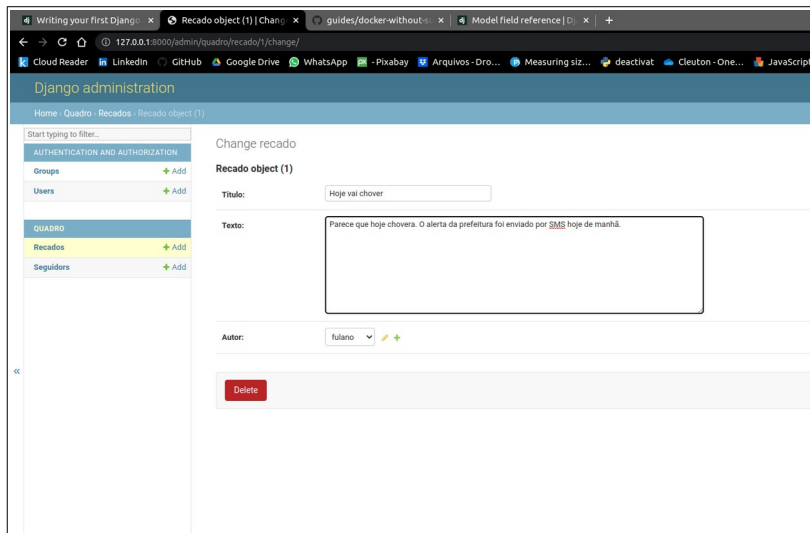


Podemos adicionar usuários, recados e seguidores. Vamos fazer isso... Adicione 2 usuários: “fulano” e “beltrano”:

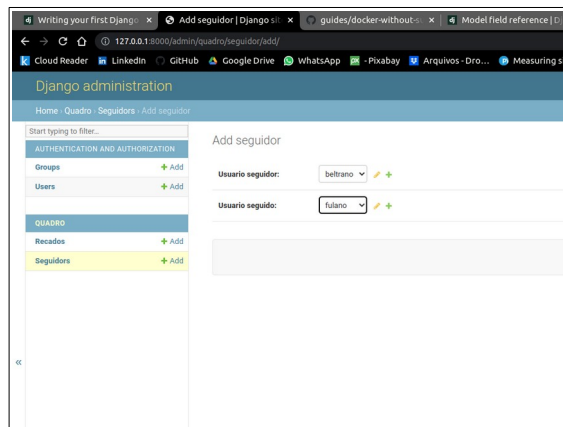


Depois, adicione 1 recado para cada 1.

Note que a chave estrangeira pode ser selecionada a partir de uma lista:



Agora, cada usuário tem 1 recado. Vamos fazer o usuário “beltrano” seguir o usuário “fulano”:



Só o usuário “beltrano” segue “fulano” e não vice-versa.

## Usando o Shell

Agora, que temos usuários e recados, podemos brincar um pouco com o Shell Python do Django, sem estarmos com o servidor no ar. Rode o comando:

```
python manage.py shell
```



Ele abrirá um shell interativo com o Python, no qual você pode importar, manipular e fazer consultas com o modelo do Django. Por exemplo, vejamos quantos recados temos:

```
>>> from quadro.models import Recado, Seguidor
>>> Recado.objects.all()
<QuerySet [<Recado: Recado object (1)>, <Recado: Recado object (2)>]>
```

Temos 2 objetos “Recado”.

Vamos exibir o primeiro recado:

```
>>> r1 = Recado.objects.get(id=1)
>>> r1.titulo
'Hoje vai chover'
>>> r1.texto
'Parece que hoje chovera. O alerta da prefeitura foi enviado por SMS hoje de manhã.'
>>> r1.data
datetime.datetime(2022, 4, 9, 12, 36, 31, 641902, tzinfo=datetime.timezone.utc)
>>> r1.autor
<User: fulano>
```

Podemos fazer CRUD com as entidades do nosso modelo. Vamos inserir um novo recado de “fulano”:

```
>>> r1.autor.id
2
```

A chave primária do autor “fulano”, e agora podemos inserir um novo recado dele:

```
>>> from django.contrib.auth.models import User
>>> u_fulano = User.objects.get(pk=2)
>>> r2 = Recado(titulo = "E pode haver ventania!", texto = "Segundo mensagem da meteorologia", autor=u_fulano)
>>> r2.save()
>>> r2.titulo
'E pode haver ventania!'
>>> r2.autor
<User: fulano>
```

Agora, como podemos pegar todos os recados de um autor? Podemos criar uma QuerySet (uma consulta) utilizando um filtro... Já temos uma instância de “autor”, que é “u\_fulano”, portanto, podemos fazer isso:

```
>>> rs = Recado.objects.filter(autor=u_fulano)
>>> for r in rs:
...     print(r.titulo)
...
Hoje vai chover
E pode haver ventania!
```

Agora, como podemos pegar os recados dos autores que seguimos? Simples:

```
>>> u_beltrano = User.objects.get(pk=3)
```

```
>>> u_beltrano
<User: beltrano>
```

Agora, pegamos todos os usuários que o “beltrano” segue:

```
>>> rs_seguindo = Seguidores.objects.filter(usuario_seguidor=u_beltrano)
```

E navegamos nessa coleção, pegando os recados dos usuários seguidos:

```
>>> for s in rs_seguindo:
...     r_seguindo = Recado.objects.filter(autor=s.usuario_seguido)
...     for r in r_seguindo:
...         print(r.titulo)
...
Hoje vai chover
E pode haver ventania!
```

Isso é basicamente o que nossa aplicação vai fazer.

Antes de continuar, eu gostaria de esclarecer algumas coisas:

1. Há mais de uma maneira de implementar essa aplicação;
2. O Django tem sempre mais de uma maneira para fazer a mesma coisa;
3. Dá para fazer mais eficiente.

Isso posto, quero dizer que sempre opto pela maneira mais simples possível e, depois que funcionar, procuro otimizar as coisas. Você está aprendendo Django, portanto, não fará muito sentido eu partir para fazer otimizações neste momento, pois você pode não compreender. Como bem o disse Donald Knuth: “premature optimization is the root of all evil”, ou “a otimização prematura é a raiz de todos os males”.

## Interface de usuário

O Django tem um mecanismo bem simples para criar UI (User Interface / Interface do Usuário) web, através das **Views** e dos **Templates**.

As Views na verdade são o equivalente ao Controller (MVC) e preparam o conteúdo que os templates exibirão. É claro que você pode gerar o HTML ou o JSON dentro do código da View, mas isso aumenta o acoplamento entre os componentes (além de dar mais uma responsabilidade à View).

Para começar, vamos criar uma view bem básica de modo a entender como funciona o mecanismo.

Abra o arquivo “quadro/views.py” e digite isso:

```
from django.http import HttpResponse

def index(request):
```

```
return HttpResponse("Bom dia")
```

Agora, precisamos mapear essa view a uma determinada URL da nossa aplicação, e isso é feito no arquivo “quadro/urls.py” (crie se não existir):

```
from django.urls import path

from . import views

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

Se você chamar a aplicação sem passar mais nada, essa é a view que será invocada.

Este é o mapeamento dentro da aplicação “quadro”, mas, se você olhar a pasta do projeto (na subpasta “recados”), verá que existe outro “urls.py” e precisamos incluir a URLconf da nossa aplicação nesse script:

“recados/recados/urls.py”:

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

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

O projeto simplesmente não sabe como acessar a nossa view. Vamos corrigir isso:

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

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

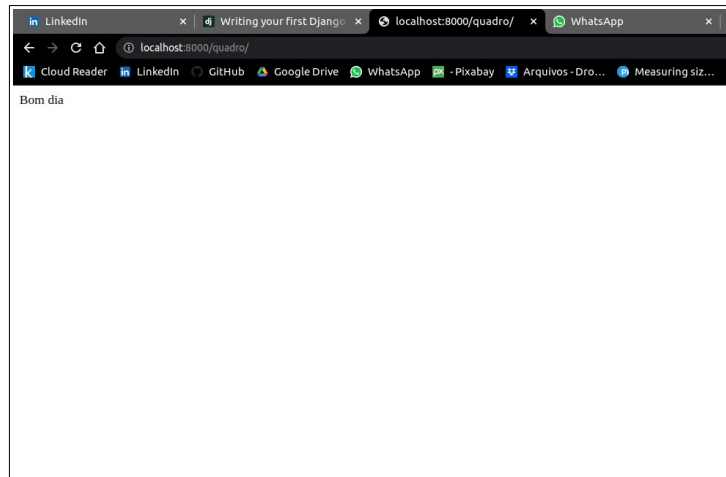
Agora, que já incluímos nosso mapeamento de URL dentro do mapeamento do Projeto, podemos rodar o servidor de desenvolvimento e invocar nossa aplicação, chamando a view “index”:

```
python manage.py runserver
```

Abra um navegador e digite:

<http://localhost:8000/quadro>

O resultado é esse:



Não parece muito animador, certo? Mas calma! Isso é só para você entender o processo de criação de views:

1. Criar o código no “views.py”:
2. Adicionar a rota no “urls.py” da aplicação;
3. Incluir o “urls.py” da aplicação no “urls.py” do projeto.

Com isso que temos agora, você já pode construir a aplicação Django. E pode fazer só REST ou até mesmo integrar com Angular.

Só que o Django tem um mecanismo de **SSR** (Server-side render) que permite gerar HTML completo para o cliente web, sem necessidade de criar frontend separado.

- *Muitos alegarão que usar SSR é coisa do passado, e que você divide melhor o processamento entre frontend e backend com um framework Javascript (Angular, React etc), porém, cabe a você avaliar a necessidade disso. Frameworks frontend são de altíssima complexidade accidental, e possuem uma longa curva de aprendizado. Em certas aplicações, páginas geradas por SSR podem funcionar muito bem, especialmente em MVPs (Produtos mínimos viáveis).*

Ok. Vamos criar uma view com template e incluir informações do banco de dados para mostrar ao usuário. É claro que precisaríamos implementar o login para saber quem é o usuário logado e mostrar os recados dele, mas, só para demonstrar o mecanismo de template, vamos fazer algo mais simples que depois mudaremos. Vamos mostrar quantos recados existem no total.

Primeiramente, vamos criar uma pasta para guardar nossos templates. Crie uma subpasta chamada “templates”, dentro da pasta da aplicação (“quadro”). E crie uma pasta “quadro” dentro dessa pasta “templates”. Depois, crie um arquivo “index.html” nessa última pasta, com esse conteúdo:

“quadro/templates/quadro/index.html”:

```
<html>
  <head>
    <title>Quadro de recados</title>
  </head>
  <body>
```

```
<p>Temos {{ quantidade_recados }} recado(s).</p>
</body>
</html>
```

Entre chaves duplas “{{” e “}}” temos a variável “quantidade\_recados”, que o template receberá através do contexto de renderização (nós passaremos para o mecanismo de SSR do Django). E temos as tags (entre “{%” e “%}”), que permitem incluirmos comandos para serem processados no momento de renderizar o HTML.

E vamos modificar nossa view:

“quadro/views.py”:

```
from django.shortcuts import render

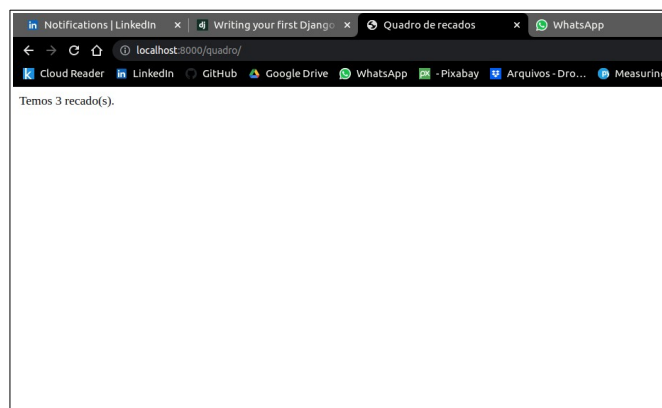
# Create your views here.

from django.http import HttpResponse
from django.template import loader
from .models import Recado

def index(request):
    quantidade_recados = Recado.objects.count()
    template = loader.get_template('quadro/index.html')
    context = {
        'quantidade_recados': quantidade_recados,
    }
    return HttpResponse(template.render(context, request))
```

É bem intuitivo o que eu fiz. Primeiro, importei o “loader” que permite carregar um template, depois, criei um dicionário de contexto, com tudo o que quero passar para o renderizador SSR do Django, finalmente, usei a instância do renderizador SSR, já com o meu template, para renderizar a resposta.

O resultado:



Ai está a página com o template.

## Autenticação e controle de acesso

Nossa aplicação se baseia em acesso identificado e não tem uma página pública. Ao acessar a aplicação, o usuário tem que se identificar, então, serão exibidos os seus recados e os recados de quem ele segue.

Você não precisa fazer quase nada para incluir controle de acesso! Essa é a beleza do Django!

O que precisamos fazer?

1. Incluir a aplicação de autenticação do Django na URLconf do projeto;
2. Criar um template de página de login;
3. Adicionar o diretório de templates no “settings.py” do projeto;
4. Adicionar uma URL de redirect de login ao “settings.py” do projeto;
5. Decorar nossa view com “@login\_required”.

Estes passos garantem que só usuários autenticados acessarão a página inicial. Vamos fazer.

Primeiro, você precisa incluir a aplicação de autenticação do Django na URLconf do projeto:

```
urlpatterns = [  
    path('quadro/', include('quadro.urls')),  
    path('admin/', admin.site.urls),  
    path("accounts/", include("django.contrib.auth.urls")),  
]
```

Crie assim mesmo, com a URL “accounts/”. Agora, seu projeto sabe como invocar essa URL da aplicação de autenticação.

Agora, vamos criar um template de página de login. Primeiramente, criamos uma subpasta “registration” sob a pasta “templates” da nossa aplicação. Então teremos essa estrutura:

```
quadro:  
  templates:  
    quadro:  
      index.html  
    registration:  
      login.html
```

Esse arquivo “login.html” deve ser assim:

```
<h2>Log In</h2>  
<form method="post">  
  {% csrf_token %}  
  {{ form.as_p }}  
  <button type="submit">Log In</button>  
</form>
```

Quem passa os dados é a aplicação de autenticação do Django. Você pode customizar isso depois.

Depois, vamos adicionar o diretório de templates no “settings.py” do projeto:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [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',
            ],
        },
    ],
]
```

E vamos adicionar uma URL de redirect de login ao “settings.py” do projeto (ao final do arquivo), desta forma, após o login, ele será redirecionado à página principal:

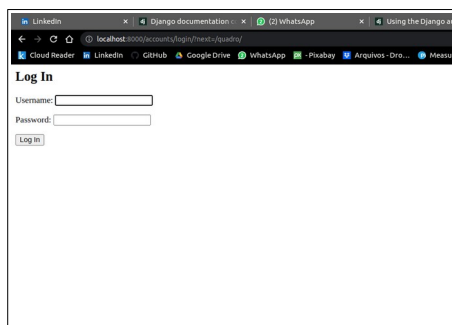
```
LOGIN_REDIRECT_URL = "/"
```

Agora, só falta decorar nossa view com “@login\_required”. Edite o arquivo “quadro/views.py” e modifique:

```
...
from django.contrib.auth.decorators import login_required

@login_required
def index(request):
    ...
```

E é só! Tente acessar a aplicação novamente e verá isso:



Após logar com qualquer um dos seus usuários (não precisa ser o super user) você verá a sua página index normalmente.

### Mas e o log out?

Sim. Você precisa incluir um link de logout, caso contrário, o usuário ficará logado muito tempo (2 semanas)! Você pode controlar isso com essa variável no “settings.py” do projeto:

```
SESSION_COOKIE_AGE=<idade em segundos>
```

E você pode disponibilizar um link de “Log out” para o seu usuário. Por exemplo, na página principal, vamos colocar isso:

```
<html>
  <head>
    <title>Quadro de recados</title>
  </head>
  <body>
    <p>Temos {{ quantidade_recados }} recado(s).</p>
    <p><a href="{% url 'logout' %}">Sair da aplicação</a></p>
  </body>
</html>
```

E vamos criar uma entrada no “settings.py” para redirecionar o usuário após o “Log out”. Vamos para a url da aplicação:

```
LOGOUT_REDIRECT_URL = "/quadro"
```

Agora, ao clicar no link “Sair da aplicação”, o usuário perde a sessão e é redirecionado para a página inicial da aplicação e, como não está mais “logado”, é redirecionado à página de login.

## Uso de sessão

O ideal é que a sua aplicação, pelo menos a parte “backend”, seja “stateless”, ou seja, sem manter estado no servidor. Mas, como você viu, estamos criando uma aplicação web monolítica para aprender Django, portanto, há um jeito de usarmos sessões e mantermos estado no servidor. Eu não vou ensinar nenhum tipo de “session backend”, só o default.

Se você abrir o “settings.py” do projeto, poderá ver os MIDDLEWARES instalados:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
```



]

Entre eles está o “SessionMiddleware”, o que nos permite salvar estado no servidor. Por exemplo, vamos alterar a view de índice para salvar a data e hora do último acesso. Primeiramente, vamos setar algumas configurações no “settings.py” do projeto:

```
USE_TZ=True
TIME_ZONE="America/Sao_Paulo"
```

Como vamos trabalhar com data e hora, eu informei ao Django que vamos usar Time zone e que o nosso time zone atual é o Brasileiro.

Em seguida, vamos alterar nosso template “index.html” para exibir mais um campo:

```
<html>
  <head>
    <title>Quadro de recados</title>
  </head>
  <body>
    <p>Último acesso: {{ ultima_data|date:"d/m/Y H:i:s" }}</p>
    <p>Temos {{ quantidade_recados }} recado(s).</p>
    <p><a href="{% url 'logout' %}">Sair da aplicação</a></p>
  </body>
</html>
```

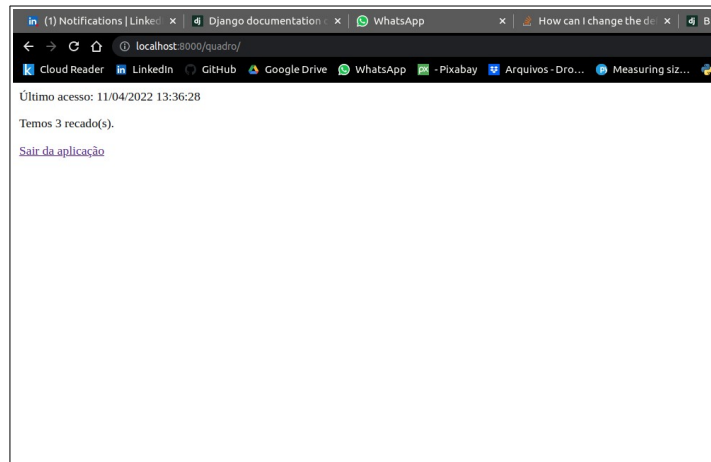
Notou o template para exibição da variável “ultima\_data”? Ele tem um filtro (caractere “|”) que formata o conteúdo da variável como data e hora, seguindo o padrão do Django (<https://docs.djangoproject.com/en/4.0/ref/templates/builtins/#date>).

E vamos modificar o código da view:

```
@login_required
def index(request):
    try:
        ultima_data = datetime.fromtimestamp(request.session['ultima_data'])
    except KeyError:
        ultima_data = datetime.now()
        request.session['ultima_data'] = datetime.timestamp(ultima_data)
    quantidade_recados = Recado.objects.count()
    template = loader.get_template('quadro/index.html')
    context = {
        'ultima_data' : ultima_data,
        'quantidade_recados': quantidade_recados,
    }
    nova_data = datetime.now()
    request.session['ultima_data'] = datetime.timestamp(nova_data)
    return HttpResponse(template.render(context, request))
```

Estamos utilizando o serializador padrão de sessão do Django, que salva tudo em JSON. Então, tudo o que desejarmos salvar tem que ser passível de serialização utilizando este formato. Datetime não é serializável em JSON, então o transformamos em “timestamp” e depois transformamos em Datetime novamente.

Agora, ao entrar na página index, ele mostrará quando foi a última vez que você entrou (e atualizará a variável de sessão):



Pronto! Eis o uso de variáveis de sessão na prática.

## Terminando a aplicação

Agora é o momento de concluirmos a aplicação. Vamos começar refazendo nossa página “index.html”.

Precisamos mostrar os recados que o usuário postou e os recados que os usuários que ele segue postaram. Vamos alterar o template para fazer isso:

```
<html>
  <head>
    <title>Quadro de recados</title>
  </head>
  <body>
    <p>Último acesso: {{ ultima_data|date:"d/m/Y H:i:s" }}</p>
    <p>Temos {{ quantidade_recados }} recado(s).</p>
    {% if seus_recados %}
      <p>Seus recados:</p>
      <ul>
        {% for recado in seus_recados %}
          <li><a href="/quadro/{{ recado.id }}">
            {{ recado.titulo }}</a></li>
        {% endfor %}
      </ul>
    {% endif %}
    {% if recados_seguidos %}
      <p>Recados de quem você segue:</p>
      <ul>
        {% for recado in recados_seguidos %}
          <li><a href="/quadro/{{ recado.id }}">
            {{ recado.titulo }}</a></li>
        {% endfor %}
      </ul>
    {% endif %}
  </body>
</html>
```

```
        {% endfor %}
    </ul>
{% endif %}

    <p><a href="{% url 'logout' %}">Sair da aplicação</a></p>
</body>
</html>
```

Deve ser bem intuitivo o que eu fiz, mas vamos lá...

Se você tiver algo na variável “seus\_recados”:

```
{% if seus_recados %}
```

Este é um tag de template. Tudo entre “{%” e “%}” é executado pelo engine de SSR do Django. Não é Python!

Então, vamos fazer um loop e pegar cada elemento da variável:

```
{% for recado in seus_recados %}
```

E vamos mostrar o título do recado e um link para exibir o recado:

```
<li><a href="/quadro/{{ recado.id }}">{{ recado.titulo }}</a></li>
```

Note que o “for” tem um “{% endfor %}” e o “if” tem um “{% endif %}”. Podemos ter “else” também.

Fizemos a mesma coisa com a variável “recados\_seguidos”.

Agora, temos que alterar o código da view para gerarmos essas novas variáveis:

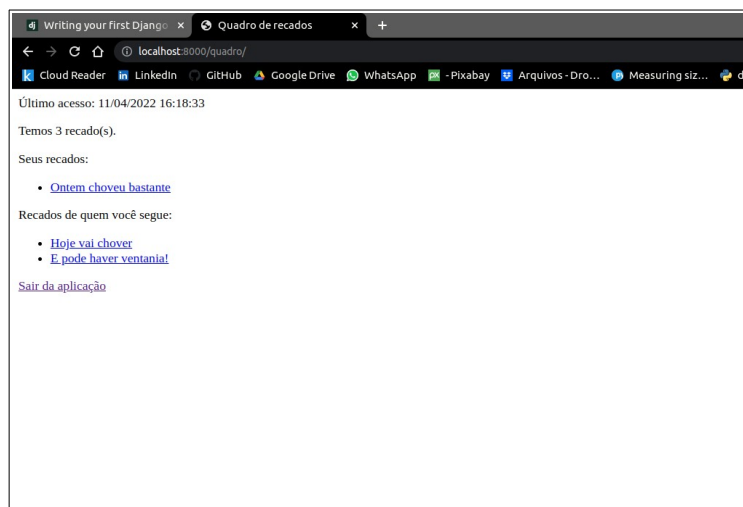
```
from django.http import HttpResponse
from django.template import loader
from .models import Recado, Seguidor
from django.contrib.auth.models import User
from django.contrib.auth.decorators import login_required
from datetime import datetime

@login_required
def index(request):
    try:
        ultima_data = datetime.fromtimestamp(request.session['ultima_data'])
    except KeyError:
        ultima_data = datetime.now()
        request.session['ultima_data'] = datetime.timestamp(ultima_data)
    quantidade_recados = Recado.objects.count()
    template = loader.get_template('quadro/index.html')

    seus_recados = Recado.objects.filter(autor=request.user)
    recados_seguidos = []
    rs_seguindo = Seguidor.objects.filter(usuario_seguidor=request.user)
    for s in rs_seguindo:
        r_seguindo = Recado.objects.filter(autor=s.usuario_seguido)
```

```
for r in r_seguindo:
    recados_seguidos.append(r)
context = {
    'seus_recados' : seus_recados,
    'recados_seguidos' : recados_seguidos,
    'ultima_data' : ultima_data,
    'quantidade_recados': quantidade_recados,
}
nova_data = datetime.now()
request.session['ultima_data'] = datetime.timestamp(nova_data)
return HttpResponse(template.render(context, request))
```

Eu só repeti as queries que fizemos anteriormente, quando eu mostrei o shell do Django para você.



Agora, só faltam algumas coisas:

- Exibir um recado;
- Incluir um novo recado;
- Excluir um recado que você postou.

Vamos criar a UI para exibir um recado, de maneira bem simples. Ao clicar em um recado na página inicial, você é direcionado por um link:

`http://localhost:8000/quadro/2/`

Então vamos criar um template para isso, chamado “detalhe.html”:

```
<html>
  <head>
    <title>Quadro de recados</title>
  </head>
  <body>
    <p>Recado: {{ recado.titulo }}</p>
    <p>Em: {{ recado.data|date:"d/m/Y H:i:s"}}</p>
    <p>Autor: {{ recado.autor.username }}</p>
    <hr>
```

```
<p>{{ recado.texto }}</p>
</body>
</html>
```

Acredito que não haja novidade nenhuma, certo? Note que o nome do autor é a propriedade “username” do usuário logado.

Eu não vou criar página para cadastrar usuários, embora poderia fazer isso.

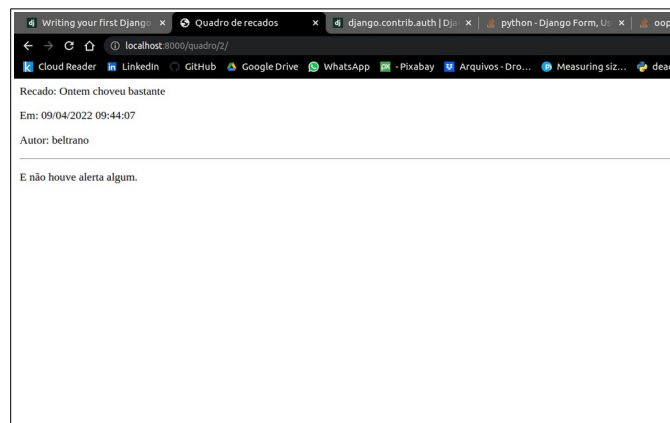
E vamos criar uma função para isso dentro de “views.py”:

```
@login_required
def detalhe(request, id_recado):
    template = loader.get_template('quadro/detalhe.html')
    recado = Recado.objects.get(id=id_recado)
    context = {
        'recado': recado
    }
    return HttpResponse(template.render(context, request))
```

Também acredito que não haja mistério algum, certo? Como é que essa função será invocada? Isso, criando uma entrada no “urls.py” da aplicação:

```
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:id_recado>/', views.detalhe, name='detalhe'),
]
```

É só rodar e clicar em um recado para vermos os detalhes:



Pronto!

Só uma coisinha que aposto que você se perguntou: E se o “recado” não existir? Então podemos gerar um erro 404. Basta alterar um pouco o código da View:

```
from django.http import Http404
...
```

```
try:
    recado = Recado.objects.get(id=id_recado)
except Recado.DoesNotExist:
    raise Http404("Recado inexistente")
```

Agora, vamos colocar um link para excluir um recado nessa mesma página, quando você for o seu autor. Para começar, vamos alterar o template “detalhe.html”:

```
...
<p>{{ recado.texto }}</p>
{% if recado.autor == logado%}
    <p><a href="/quadro/excluir/{{ recado.id }}">Excluir recado</a></p>
{% endif %}
</body>
```

Se você for o autor do recado, então aparecerá um link para você excluir o mesmo.

E temos que criar uma nova função na view:

```
from django.core.exceptions import PermissionDenied
from django.shortcuts import redirect
```

```
@login_required
def excluir(request, id_recado):
    try:
        recado = Recado.objects.get(id=id_recado)
    except Recado.DoesNotExist:
        raise Http404("Recado inexistente")
    if request.user != recado.autor:
        raise PermissionDenied()
    recado.delete()
    return redirect('index')
```

Tive que importar mais duas coisas: A exception de “PermissionDenied” e a função “redirect”. Após excluir o recado eu mando um HTTP Redirect para a view “index”. Se você forçar a barra e tentar excluir um recado que não é seu, tomará um erro 403 – Forbidden.

Só falta criar essa rota no “urls.py”:

```
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:id_recado>/', views.detalhe, name='detalhe'),
    path('excluir/<int:id_recado>/', views.excluir, name='excluir'),
]
```

Para terminar, vamos criar a página de inclusão de recados. Para isso, vamos utilizar um form. Primeiramente, criamos um novo template: “incluir.html”:

```
<html>
  <head>
    <title>Quadro de recados</title>
  </head>
  <body>
    <form action="{% url 'incluir' %}" method="post">
```

```
{% csrf_token %}
<fieldset>
    {% if error_message %}<p><strong>
        {{ error_message }}
    </strong></p>{% endif %}
    <label for="titulo">Título:</label><br>
    <input type="text" name="titulo" id="titulo" />
    <br>
    <label for="texto">Texto:</label>
    <br>
    <textarea name="texto" id="texto" rows="10"
        cols="50"></textarea>
</fieldset>
<input type="submit" value="Cadastrar">
</form>
</body>
</html>
```

Nenhuma novidade, certo? Bem... Algumas. Para começar, temos a “action” do form, que deve usar o comando “url” e o nome da view (definida no “urls.py” que receberá o POST). E temos esse “csrf\_token”... Sabe o que é?

- *Cross-site request forgery (CSRF), também conhecida como ataque de um clique ou sessão de equitação e abreviada como CSRF ou XSRF, é um tipo de exploração maliciosa de um site em que comandos não autorizados são enviados de um usuário em quem o aplicativo da web confia.*

Sempre que você tiver um form que utilize POST, um método considerado “inseguro”, você pode incluir o csrf\_token para prevenir ataques CSRF. Mas isso não pode ser feito se o POST for para uma URL externa à sua aplicação! Cuidado!

Temos uma variável “error\_message” que não estamos usando, mas podemos deixar aí para futuras validações.

Ok, temos que ter uma rota para exibir esse formulário, então vamos alterar nosso “views.py” para incluir a função:

```
@login_required
def form_incluir(request):
    template = loader.get_template('quadro/incluir.html')
    context = {}
    return HttpResponse(template.render(context, request))
```

Nada demais, certo? Mas temos que criar a rota para essa função dentro de “urls.py”:

```
path('form_incluir/', views.form_incluir, name='form_incluir'),
```

E podemos incluir um link na página “index.html”:

```
<p><a href="{% url 'form_incluir' %}">Incluir novo recado</a></p>
```

Agora, precisamos criar a função que processa a inclusão. Já indicamos a URL no template (“incluir”) agora, vamos criar a função em “views.py”:

```
@login_required
def incluir(request):
    recado = Recado(titulo=request.POST['titulo'], \
                    texto=request.POST['texto'], \
                    autor=request.user)

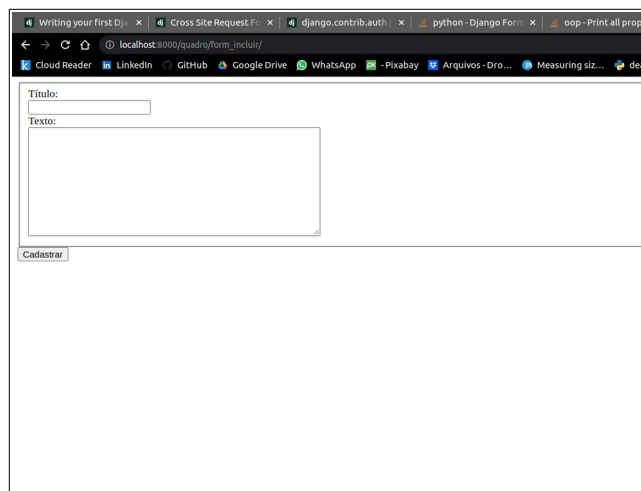
    recado.save()
    return redirect("index")
```

A única diferença para outras views é que estamos pegando o “título” e o “texto” do dicionário “POST” do “request”, pois eles foram enviados em um form.

Para essa função de incluir ser invocada, temos que incluí-la no “urls.py”. Nosso “urls.py” final ficará assim:

```
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:id_recado>/', views.detalhe, name='detalhe'),
    path('excluir/<int:id_recado>/', views.excluir, name='excluir'),
    path('form_incluir/', views.form_incluir, name='form_incluir'),
    path('incluir/', views.incluir, name='incluir')
]
```

Agora é só testar! Se você clicar no link da página “index.html”, o form será exibido:



Ao clicar em “Cadastrar”, o recado será inserido no banco de dados e a aplicação será redirecionada para a página “index.html”, onde o novo recado aparecerá.

## Conclusão



Se você conseguiu fazer todo este tutorial, então parabéns! Criou uma aplicação Python/Django utilizando PostgreSQL.

Ah, mas está muito “feia”, sim, está! O objetivo não era beleza, mas sim: Aprendizado.

Pessoas mais experientes em Django diriam para usar generic views, ou mesmo Django REST Framework. Eu digo que, antes de aprender a correr, você precisa aprender a andar. Acostume-se a fazer as coisas mais simples e você sentirá a necessidade de otimizar futuramente.

É claro que ficou de fora a parte de testes, mas isso é assunto para outro tutorial. Testes com Django são muito simples e você pode ver isso nesse site:

<https://docs.djangoproject.com/en/4.0/intro/tutorial05/>

O Django tem um tutorial muito legal, porém eles cometem os mesmos erros que todos os criadores de frameworks:

1. Só fazem em inglês;
2. Ficam otimizando enquanto estão ensinando.

O maior problema é a otimização enquanto ensina. Eles mostram uma maneira, depois otimizam, depois otimizam novamente e vão otimizando até o final. Se você ainda está aprendendo, fica com muitas dúvidas.

Próximos passos:

1. Estude e crie alguns testes automatizados, conforme eu já mencionei;
2. Estude Generic Views: <https://docs.djangoproject.com/en/4.0/topics/class-based-views/generic-display/>
3. Estude o Django REST Framework: <https://www.django-rest-framework.org/>;
4. Estude Django (Django REST Framework) com Angular: <https://medium.com/swlh/django-angular-4-a-powerful-web-application-60b6fb39ef34>;

Eu estou preparando mais tutoriais sobre Django e vou mostrar em breve:

- Django REST Framework + Angular;
- Django + GraphQL com Graphene-Django;
- Django REST Framework + JWT com Simple JWT.

## Devo utilizar Django?

Essa é a pergunta de 1 milhão de dólares! Eu gosto muito do que o Robert C. Martin diz sobre usar ou não um framework:

- *Nunca compre magia! Antes de se comprometer com um framework, certifique-se de poder escrevê-lo. Faça isso escrevendo algo simples que faça o básico que você precisa.*

*Certifique-se de que toda a magia vá embora. E então olhe para o framework novamente. Vale a pena? Você pode viver sem isso?*

Robert C. Martin, “Make the Magic go away”, *The Clean Code Blog*  
<http://blog.cleancoder.com/uncle-bob/2015/08/06/LetTheMagicDie.html>

Um framework como o Django pode facilitar sua vida, no início da aplicação, quando você está escrevendo muito *boilerplate code*, mas e ao longo do ciclo de vida do software que você está fazendo? Quantas vezes você terá que reescrever rotinas básicas, como: Acessar um banco de dados com SQL, ou mudar rotas REST escritas com Flask? Poucas vezes! A maioria das vezes você vai mexer no código funcional.

Mas você “engessou” sua aplicação com o framework! Sim! Você e sua empresa estão presos ao *roadmap* do criador do framework, sob pena de deixar de receber correções de segurança!

Será que vale a pena essa “prisão” só por conta de menos de 1% da vida útil da aplicação? Não seria melhor usar uma boa ferramenta ou projeto de *scaffolding* para gerar o código para você?

O maior problema é que os cursos e faculdades já ensinam a usar os frameworks. Os alunos não aprendem Python, aprendem Django! E saem replicando esse modelo indiscriminadamente. E isso não é só com o Django, mas com todos os frameworks (Spring, Angular etc).

Então, como os Americanos dizem, não existe “almoço grátis”. Sempre que alguém lhe oferecer vantagem, pode acreditar que vão querer algo em troca, e, neste caso, é a fidelidade a um *roadmap* que você desconhece e não controla.

Meu conselho é o mesmo do Robert C. Martin: Certifique-se de que consiga fazer sem usar o Django. Se conseguir, pelo menos uma prova de conceito, então avalie se valerá a pena utilizá-lo.