Fazendo o deploy de uma aplicação Django

Yan Orestes

25/12/2018

[Formação DevOps](https://www.alura.com.br/formacao-devops)

Sabemos como o Django, framework web escrito em Python, pode ser bom para aplicações web. Mas ao final de todo o desenvolvimento, o que fazemos com todo o projeto?

Recentemente, desenvolvi um [simples projeto de agenda de contatos](https://github.com/yanorestes/aluraagenda) utilizando o framework [**Django**](https://www.djangoproject.com/), e resolvi compartilhar pela Internet. É uma aplicação web, então o ideal seria que fosse possível acessá-la como acessamos qualquer outro site, digitando seu endereço em um navegador qualquer, em um computador qualquer.

Assim, me preparei aluguei um [**VPS**](https://pt.wikipedia.org/wiki/Servidor_virtual_privado) com Linux e registrei um domínio estilo dominio.com.br. O serviço do servidor possuía diversos tutoriais ensinando a colocar projetos em PHP e Java de pé, mas não de Django… E aí, como podemos rodar nossa aplicação?

**Enviando os arquivos locais para o servidor remoto**

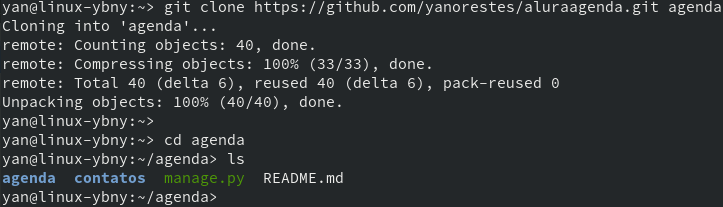
A primeira coisa que precisamos fazer é, realmente, mandar nossa aplicação para o servidor que eu aluguei. Afinal, atualmente tenho todos os meus arquivos aqui no meu computador, mas não vou rodar a aplicação por ele.

Para esse primeiro passo, temos várias opções, como [**FTP**](https://pt.wikipedia.org/wiki/File_Transfer_Protocol), [**SFTP**](https://pt.wikipedia.org/wiki/SSH_File_Transfer_Protocol), ou o [**Git**](https://www.alura.com.br/artigos/comecando-com-git-aprendendo-versionar/). Por conta da simplicidade, vou optar pelo Git, aproveitando que meu projeto já está no [GitHub](https://github.com/yanorestes/aluraagenda).

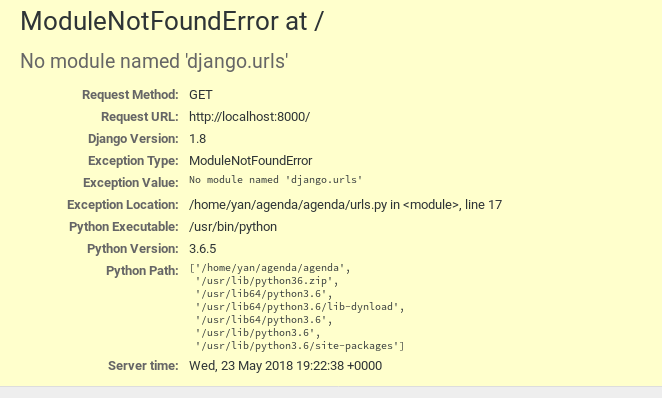
Em minha pasta de usuário (home), quero clonar (clone) o repositório que está lá no GitHub para um diretório chamado agenda. Um simples comando faz o trabalho:

git clone https://github.com/yanorestes/aluraagenda.git agenda

Rapidamente, podemos ver que uma pasta agenda (como especificamos no final do comando) foi criada em nosso diretório home, contendo os arquivos do projeto - a aplicação Django em si:



Como a maioria dos sistemas Linux, o meu já veio com o Python instalado. Além disso, fiz um pedido para que viesse também com o Django. Assim, fui tentar rodar a aplicação, como teste, mas olha o que aconteceu:



Mas, ué, na minha máquina local estava funcionando perfeitamente! O que foi que deu? Com um pouco de atenção, eu percebi - a mensagem de erro indicava que a versão do Django era a 1.8.

O problema é que eu programei me baseando na versão 2.0, que é mais recente e quebra algumas coisas das versões passadas. Será que vou ter que desinstalar o Django e instalar a nova versão?

Isso até poderia resolver… Se bem que eu tenho alguns outros projetos em que uso a versão 1.8 do Django, e estava até pensando em deixar nesse servidor. Mas e agora? Pelo visto, o ideal seria ter isso mais separado, sem cada instalação interferir de um jeito diferente o nosso projeto. E como podemos fazer isso?

**Criando um ambiente virtual Python com o *virtualenv***

Queremos dividir nossas aplicações em ambientes separados, cada qual com suas especificidades, para não nos embaralharmos com nossos pacotes Python.

Ou seja, queremos criar ambientes virtuais (***virtual environments***), onde cada ambiente terá sua própria versão dos pacotes. Mas como podemos criá-los?

No Python, temos o [***virtualenv***](https://virtualenv.pypa.io/en/stable/), que cria ambientes Python isolados, nos quais podemos aplicar as configurações que desejarmos.

Podemos instalá-lo através do [**pip**](https://www.alura.com.br/artigos/como-publicar-seu-codigo-python-no-pypi/), com o seguinte comando no terminal:

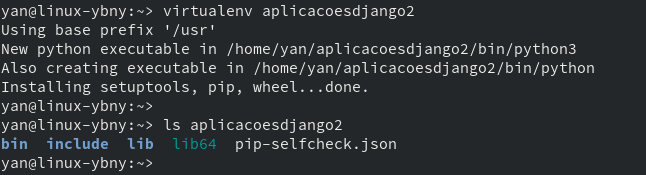
pip install virtualenv

*Dependendo da configuração do seu sistema, será necessária permissão sudo para rodar esse comando*

Com o *virtualenv* instalado, podemos criar um novo ambiente com um simples comando, especificando o nome que queremos definir:

virtualenv aplicacoesdjango2

Rodando esse comando, um diretório deve ser criado:



Precisamos, agora, dizer para o terminal que queremos entrar nesse ambiente, isto é, ativar esse *virtualenv* que criamos. Fazemos isso com o comando source, especificando a localização do arquivo de ativação, que é sempre a mesma:

source aplicacoesdjango2/bin/activate

E olha como fica o terminal agora:

O terminal muda quando estamos dentro de uma virtualenv

Podemos, enfim, instalar a versão que queremos do Django para esse ambiente virtual, através do pip:

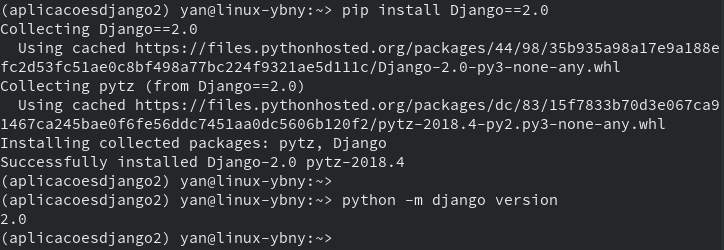
pip install Django==2.0

*É importante que esse comando não seja executado com sudo, ou o pip fará a instalação no Python local da máquina, não do ambiente virtual.*

Podemos verificar se a instalação foi bem sucedida com um comando com o Python:

python -m django version

Se tudo der certo, o resultado deve indicar **2.0**:



Para finalizar essa parte, vamos mover (mv) nossa aplicação para o diretório do ambiente virtual, apenas para manter uma melhor organização:

mv agenda aplicacoesdjango2

Certo! Entrei (cd) no diretório da aplicação para testar:

cd aplicacoesdjango2/agenda

Coloquei o servidor do Django para rodar, com o clássico comando runserver:

python manage.py runserver

Mas… novamente um erro! Dessa vez, direto no terminal:

*## ERRO OMITIDO*

django.core.exceptions.ImproperlyConfigured: Error loading MySQLdb module.

Did you install mysqlclient?

A longa exceção que recebemos indicou um erro de configuração, nos questionando se instalamos o driver conector do MySQL com o Python. De fato, não instalamos. Na verdade, nem ao menos instalamos o MySQL em si! Vamos cuidar disso, então.

**Instalando o MySQL (MariaDB)**

Antes de tudo, precisamos apertar **CTRL-C** para sair do runserver falho. Feito isso, agora precisamos instalar o banco de dados.

No caso, usarei o [**MariaDB**](https://mariadb.org/), que é um *fork* do MySQL, porque é como está configurado em minha máquina local. Caso quiséssemos usar outro serviço, como o PostgreSQL, ou o SQLite, essa parte de configuração seria diferente.

A instalação também depende da distribuição Linux e gerenciador de pacotes que estamos usando. No nosso caso, vamos focar na instalação com o [**apt**](https://pt.wikipedia.org/wiki/Advanced_Packaging_Tool), padrão de distribuições derivadas do Debian, como o Ubuntu, por ser o mais conhecido.

Apesar disso, é importante que fique claro que não é a única possibilidade! Bastaria um pouco de pesquisa para descobrir os nomes dos pacotes relativos ao que instalaremos pelo apt.

Antes de começar a instalação, vamos sair do nosso ambiente virtual, pois não precisamos dele agora. Para isso, basta usar o comando deactivate no terminal:

Saindo da virtualenv com o deactivate

Partindo para a instalação, temos quatro pacotes no apt que precisamos instalar - mariadb-server, python-dev, libmariadbclient-dev, libssl-dev. Esses pacotes contêm, além do banco de dados, arquivos de cabeçalho importantes.

Instalados os pacotes, vamos garantir que o MariaDB já está rodando:

sudo service mysql start

Agora podemos partir para a instalação do driver de conexão com o Python. Como é um pacote Python, vamos voltar para nosso *virtualenv*:

source ~/aplicacoesdjango2/bin/activate

O pacote, de nome mysqlclient, pode ser instalado com o pip:

pip install mysqlclient

Tudo ok! Com tudo instalado, precisamos configurar nosso banco, criando a *database*.

**Configurando o MySQL (MariaDB)**

Temos o banco de dados e suas dependências instaladas, mas ele ainda está completamente vazio, sem configuração alguma.

Assim, antes de qualquer coisa, é importante que criemos nossa database (CREATE DATABASE) agenda e nosso usuário (CREATE USER), que vou chamar de yan. Também temos que dar os privilégios necessários (GRANT ALL PRIVILEGES) para nosso usuário.

*Caso SQL ainda seja um tópico confuso, temos uma*[*apostila disponível gratuitamente*](https://www.alura.com.br/apostilas)*que trata da linguagem e de bancos de dados relacionais, além dos diversos*[*cursos na Alura*](https://www.alura.com.br/carreira-iniciando-com-sql-e-mysql)*! ;)*

Note que ainda não criamos a tabela de contatos que meu projeto usa, e é porque fazer isso manualmente é desnecessário! Já já vamos ver como o Django pode nos ajudar nisso.

Com o básico do banco configurado, precisamos passar para o próprio Django essas configurações. Para isso, vamos alterar algumas coisas no arquivo setttings.py, dentro da pasta agenda, de nossa aplicação agenda. Vamos entrar em nosso ambiente virtual e editar o arquivo com o nano:

source ~/aplicacoesdjango2/bin/activate

nano ~/aplicacoesdjango2/agenda/agenda/settings.py

E agora, o que devemos mexer nele?

**Configurando o banco de dados no Django**

Com o arquivo settings.py aberto, vamos procurar a parte que trata do banco de dados, localizada no dicionário DATABASES. Dentro dele, há outro dicionário, 'default', com as configurações padrões - é nele que queremos mexer.

A primeira variável, 'ENGINE', diz respeito ao tipo de banco de dados que estamos usando. Como estamos com o MariaDB, que é um fork do MySQL, devemos colocar o valor 'django.db.backends.mysql'. 'NAME' trata do nome da *database*, e USER e PASSWORD das informações do usuário. Podemos deixar 'HOST' e 'PORT' vazios, já que o banco está rodando no próprio [localhost](https://pt.wikipedia.org/wiki/Localhost) do servidor. Ao final, esse bloco do código ficou assim:

DATABASES = {

'default': {

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

'NAME': 'agenda',

'USER': 'yan',

'PASSWORD': 'senha',

'HOST': '',

'PORT': '',

}

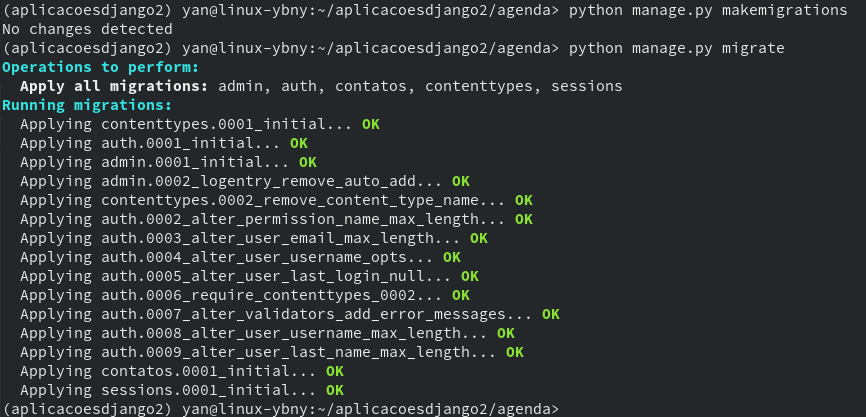
}

Com essa estrutura definida, podemos fazer a migração com o Django:

python manage.py makemigrations

python manage.py migrate

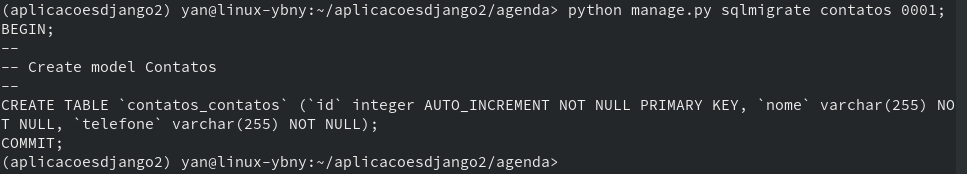
Algumas mensagens devem aparecer indicando que tudo deu certo:



Só falta criar a tabela que configurei no meu projeto Django, a contatos. Aqui, o Django pode nos ajudar muito com um simples comando, no qual só precisamos passar o nome do *app* e o nome da migração:

python manage.py sqlmigrate contatos 0001

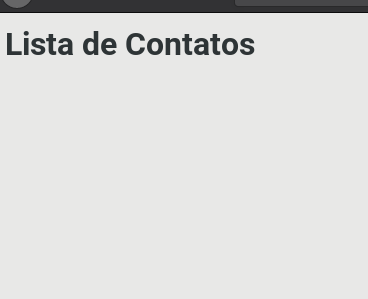
Olha só:



Com a tabela criada, vamos rodar o servidor com o runserver para ver se dessa vez dá certo:

python manage.py runserver

E agora, quando testo no navegador, através da porta **8000** (padrão do Django):



Certo! Bem, já conseguimos até acessar o site, mesmo que por uma porta que não é padrão do HTTP. Então é isso, está tudo certo? Qual seria o problema de deixar dessa forma?

**As questões do servidor de desenvolvimento do Django**

Logo quando comecei a me aventurar no mundo do Django, já percebi que era um consenso não usar o servidor de desenvolvimento nativo do framework em ambiente de produção. Na verdade, [**a própria documentação**](https://docs.djangoproject.com/en/2.0/intro/tutorial01/#the-development-server) da ferramenta nos indica isso. Mas por quê, se nos meus testes ele funciona bem?

O que acontece é que o ponto é justamente esse - testes. Esse servidor do Django existe apenas como uma forma rápida e otimizada de rodar nosso código durante o desenvolvimento. Assim, possui algumas características que o tornam impróprios para uso em produção.

Em primeiro lugar, o servidor do Django **não é seguro**. E por quê? Pelo simples fato de não ter sido criado para isso - segurança do servidor não é uma preocupação enquanto queremos testar apenas o código da aplicação; é um setor separado.

Outro ponto está nas *features* que esse servidor carrega. Além de não suportar algumas que podem ser bastante importantes, como *multi-threading*, tem outras, como a habilidade de aplicar as mudanças que fazemos sem termos de recarregar o servidor manualmente, que podem forçar bastante o processamento da máquina.

Finalmente, temos a questão da performance em si. O servidor do Django simplesmente não aguenta muitas requisições ao mesmo tempo, o que pode causar lentidão, travamentos e até quedas.

Por conta desses motivos, não devemos utilizar o servidor de desenvolvimento disponível pelo Django para nada além do… desenvolvimento! Ele é incrivelmente útil enquanto programamos, mas não é considerável em ambiente de produção. Mas se é assim, como devemos estruturar nosso servidor remoto?

**O fluxo de conexão de uma aplicação Django**

No geral, qualquer aplicação web precisa de um serviço de servidor responsável por atender as requisições [HTTP](https://www.alura.com.br/artigos/qual-e-diferenca-entre-http-e-https/), respondendo com conteúdo estático e/ou dinâmico, se integrando com outros serviços. Temos muitos exemplos desse tipo de servidor web, como o [**Apache**](https://httpd.apache.org/), o mais conhecido, e o [**Nginx**](https://nginx.org/en/), também muito utilizado. No nosso caso, usaremos o **Nginx**, por ser leve e rápido.

O Nginx vai funcionar muito bem para requisições de imagens e outros arquivos estáticos. Mas e nossa aplicação Django em si? Precisamos de algo para executar todo o código em Python, e esse tipo de servidor web não consegue fazer isso. A resposta mais óbvia talvez seja usar o próprio Python puro para isso, mas ele também não é o melhor para lidar com todos os tipos de requisições. E aí?

Assim, para requisições que precisam ser geradas dinamicamente, usamos, geralmente, um servidor de aplicação. O Nginx, caso necessário, vai passar a requisição para esse servidor, que, lidando com ela, vai devolver uma resposta ao servidor web, que, finalmente, responderá diretamente ao cliente.

Esses dois servidores se comunicam, por padrão, através de uma interface especificada no [**PEP 333**](https://www.python.org/dev/peps/pep-0333/), denominada [**WSGI**](https://pt.wikipedia.org/wiki/Web_Server_Gateway_Interface). Existem diversos serviços de servidores de aplicação que suportam essa interface, como o [mod\_wsgi](https://modwsgi.readthedocs.io/) e o [**Gunicorn**](http://gunicorn.org/), que será o que usaremos, por ser bem rápido e escrito puramente em Python. Vamos começar cuidando do Nginx, para então partir para o Gunicorn.

**Instalando e configurando o Nginx**

Antes de tudo, precisamos instalar o Nginx em nossa máquina. Para isso, podemos desativar ( deactivate) nossa *virtualenv*. Instalar o pacote é fácil com o apt:

sudo apt-get install nginx

*Em sistemas que não utilizam do apt, esse processo também não deve ser nada difícil, provavelmente apenas usando um comando diferente.*

Com o Nginx instalado, podemos logo inicializá-lo, como fizemos com o MySQL:

sudo service nginx start

Agora podemos passar para a parte de configuração. Precisamos explicar para o Nginx como queremos que ele lide com as requisições.

Para isso, vamos criar um arquivo de configuração específica para nosso projeto no diretório sites-available do Nginx. Vamos nos mover para esse diretório com o comando cd:

cd /etc/nginx/sites-available

Utilizando o apt, provavelmente conseguimos entrar no diretório que queríamos, porque o pacote do Nginx mantido por lá automaticamente vem com as pastas sites-available e sites-enabled. Entretanto, em outros sistemas é possível que isso resulte em um erro indicando que essa pasta não existe. Se for o caso, a resolução é simples - podemos criar esses dois diretórios manualmente:

sudo mkdir /etc/nginx/sites-available

sudo mkdir /etc/nginx/sites-enabled

Com as duas pastas criadas, precisamos abrir o arquivo de configuração geral do Nginx:

sudo nano /etc/nginx/nginx.conf

Agora basta adicionar uma linha **dentro do bloco http { }**:

include /etc/nginx/sites-enabled/\*;

Agora sim (caso essas pastas não tenham sido criadas automaticamente), podemos entrar na pasta sem problemas:

cd /etc/nginx/sites-available

Vamos criar nosso arquivo de configuração do projeto, então:

sudo nano agenda

Nosso arquivo de configuração de projeto do Nginx, que será incluído dentro do bloco http { } nas configurações gerais, tem uma estrutura básica, com um bloco server {}. Dentro desse bloco, vamos definir a variável server\_name para especificar a que sites estamos tratando:

server {

server\_name meudominio.com.br www.meudominio.com.br;

}

*meudominio deve ser alterado para o domínio que registramos e/ou o IP do servidor*

Além disso, vamos adicionar um bloco location { } **dentro do nosso bloco server { }**, especificando como o Nginx deve responder para requisições em localizações específicas. No nosso caso, queremos que para qualquer localização a partir da raiz /, a requisição seja encaminhada para o Gunicorn.

Ainda não configuramos o Gunicorn, mas podemos definir agora mesmo que queremos que ele rode no próprio localhost (127.0.0.1) na porta **8000**. Assim, adicionamos o seguinte ao bloco server { }:

location / {

proxy\_pass http://127.0.0.1:8000;

}

Legal! O único problema é que quando o Nginx enviar a requisição para o Gunicorn, o endereço de IP e o hostname específicos dessa requisição se tornarão o localhost e o hostname do nosso servidor. Para resolver isso, basta adicionar algumas linhas no bloco location / { }. Ao final, o arquivo fica assim:

server {

server\_name meudominio.com.br www.meudominio.com.br;

location / {

proxy\_pass http://127.0.0.1:8000;

proxy\_set\_header X-Forwarded-Host $server\_name;

proxy\_set\_header X-Real-IP $remote\_addr;

add\_header P3P 'CP="ALL DSP COR PSAa PSDa OUR NOR ONL UNI COM NAV"';

}

}

Certo! Temos a configuração de nosso site disponível. Precisamos avisar o Nginx que este site não só está disponível, como também ativo, e para isso vamos para o diretório sites-enabled:

cd /etc/nginx/sites-enabled

Dentro da pasta, vamos criar um link com a configuração que acabamos de criar, usando o comando ln no terminal:

sudo ln -s /etc/nginx/sites-available/agenda

Se nosso Nginx já veio com a pasta sites-enabled, provavelmente há um arquivo padrão que não queremos no diretório. Para removê-lo, basta o comando rm:

sudo rm default

Ao final, podemos reiniciar o Nginx, para aplicar as mudanças, com o comando:

sudo service nginx restart

Agora, podemos partir para o Gunicorn.

**Instalando e configurando o Gunicorn**

Com o Gunicorn, as coisas ficam bem mais simples. Primeiramente, por ser um pacote Python, podemos ir lá para o diretório de nossa aplicação e ativar o *virtualenv*:

cd ~/aplicacoesdjango2/agenda

source ../bin/activate

Para instalar o Gunicorn, basta usar o pip:

pip install gunicorn

Agora já podemos iniciá-lo. É importante que lembremos de usar o mesmo endereço de IP e a mesma porta que especificamos lá para o Nginx:

gunicorn -b 127.0.0.1:8000 agenda.wsgi

Finalmente, podemos acessar nosso site pelo navegador, através da porta padrão do HTTP. Para testar nossa aplicação de agenda de contatos, resolvi tentar adicionar alguns contatos através da página de **admin** (http://meudominio.com.br/admin):



Ué! Que página feia! Quando testei no meu próprio computador não ficou assim… Dei uma olhada no meu terminal rodando o Gunicorn e olha o que apareceu:

Not Found: /static/admin/css/base.css

Not Found: /static/admin/css/login.css

Not Found: /static/admin/css/responsive.css

Aparentemente os arquivos estáticos de CSS que personalizavam essa página não foram encontrados na URL /static/admin/css/.... De fato, esses arquivos não existem (ainda!), muito menos nessas URLs. E agora?

**Coletando e configurando os arquivos estáticos**

O Django tem, por padrão, alguns arquivos estáticos que melhoram a usabilidade padrão do framework. Coletá-los é simples, podemos fazer com apenas um comando com o manage.py. Vamos fechar o Gunicorn com **CTRL-C** e cuidar disso, então.

Antes, porém, o Django precisa saber onde queremos guardar esses nossos arquivos. Eu decidi deixar em uma pasta static/, fora da aplicação Django, mas dentro do ambiente virtual.

Além disso, quero que qualquer arquivo dentro desta página apareça como na URL meudominio.com.br/static/. Assim, modifiquei as últimas linhas do meu arquivo agenda/settings.py e deixei da seguinte forma:

STATIC\_ROOT = '/home/yan/aplicacoesdjango2/static/'

STATIC\_URL = '/static/'

Agora sim, podemos simplesmente rodar o comando no terminal:

python manage.py collectstatic

Rapidamente, os arquivos vão ser todos copiados:

118 static files copied to '/home/yan/aplicacoesdjango2/static'.

Certo! Agora já temos os arquivos que precisamos. Mas espera… como o Nginx vai saber que quando a requisição for para a URL meudominio.com.br/static/\* ele deve buscar nessa pasta que criamos? Precisamos avisá-lo!

Vamos voltar para a edição de nosso arquivo de configuração do projeto do Nginx:

sudo nano /etc/nginx/sites-available/agenda

Agora, dentro do bloco server { }, vamos precisar de mais um bloco location { } específico para as URLs /static/, e que funcione como um atalho para nossa pasta lá no ambiente virtual. Assim, podemos adicionar isso no arquivo:

location /static/ {

alias /home/yan/aplicacoesdjango2/static/;

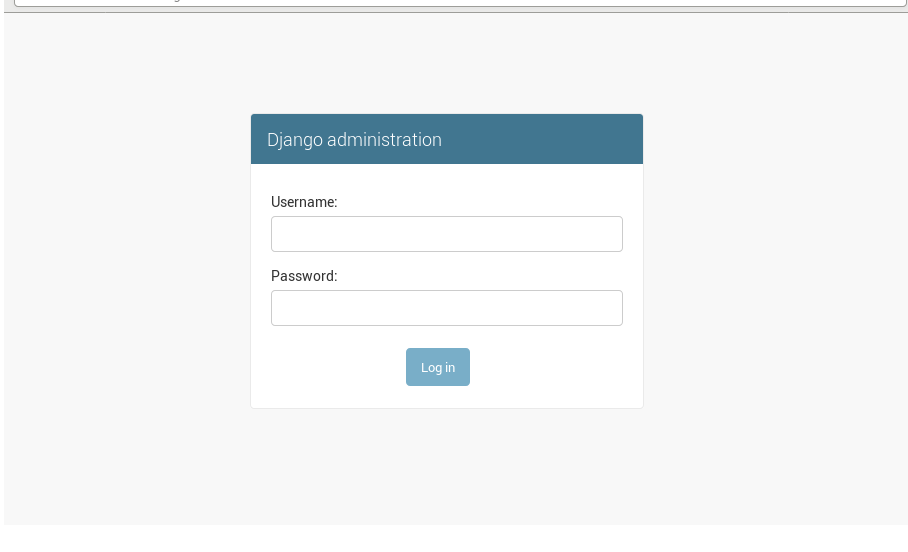
}

Feito! Podemos reiniciar o Nginx e executar novamente o Gunicorn:

sudo service nginx restart

gunicorn -b 127.0.0.1:8000 agenda.wsgi

Dessa vez, olha como ficou a página de admin:



Muito mais bonito, com o CSS carregado! Agora é só digitarmos nosso usuário e senha e… mas quando definimos isso tudo? Ainda não configuramos nosso usuário de administrador da aplicação! Para isso, podemos fechar o Gunicorn e rodar um simples comando:

python manage.py createsuperuser

Seguindo as instruções, tudo deve estar certo!

Apenas para finalizar, a **última** coisa que não podemos esquecer é de **alterar o valor da variável DEBUG para False no agenda/settings.py, agora que estamos a nível de produção**. Assim, podemos rodar novamente o Gunicorn que tudo funcionará como deve!

**Trabalhando com o Django com clareza**

Fazer o deploy de uma aplicação Django não precisa ser estressante, como vimos hoje! Apesar do Django ser, em geral, menos utilizado que certas outras ferramentas, ele continua sendo incrivelmente poderoso, como evidenciam sites como o [Instagram](https://thenewstack.io/instagram-makes-smooth-move-python-3/). Assim, usá-lo (e usá-lo bem) pode ser uma ótima opção para o seu projeto!

Após toda a programação, chega sempre um dos momentos mais temidos para os desenvolvedores - o deploy. Apesar dos mitos, hoje aprendemos passo a passo como colocar no ar nossa aplicação Django, de forma simples e eficiente. Daqui para frente não teremos mais esse problema!

Além disso que aprendemos hoje, é bacana saber que temos mais opções que podem nos ajudar com o deploy de aplicações, como o [**Docker**](https://www.alura.com.br/curso-online-docker-e-docker-compose).

Se quiser continuar estudando sobre o assunto, dê uma olhada em nossos cursos na Alura de [**Django**](https://www.alura.com.br/curso-online-introducao-ao-django) e [**Python**](https://www.alura.com.br/cursos-online-programacao/python) e continue aprendendo!

Leia também: