Sumário

[**Ambiente de desenvolvimento** 1](#_Toc118639400)

[**Virtualenv** 1](#_Toc118639401)

[**Servidor** 2](#_Toc118639402)

[**Idioma e Timezone** 3](#_Toc118639403)

[**Variáveis de ambiente** 3](#_Toc118639404)

[**Git e Github** 4](#_Toc118639405)

# Ambiente de desenvolvimento

Nesse treinamento, vamos utilizar o [**VSCode**](https://code.visualstudio.com/download) como editor de código e o [**GitHub**](https://github.com/) para o versionamento de código.

Será necessário o instalador de pacotes **pip**. Abra o CMD/prompt ou terminal, conforme o seu sistema operacional e use os seguintes códigos:

Para sistemas Windows:

curl [https://bootstrap.pypa.io/get-pip.py -o get-pip.py](https://bootstrap.pypa.io/get-pip.py%20-o%20get-pip.py)

python get-pip.py

Ao decorrer do curso iremos utilizar um ambiente virtualizado chamado [**virtualenv**](https://github.com/pypa/virtualenv) para construir os projetos da aula. Você pode instalar a **virtualenv** com o seguinte comando:

pip install virtualenv

# Virtualenv

Agora vamos isolar todas as dependências do projeto. Essa é uma boa prática de programação.

Faremos isso com o comando virtualenv venv. Depois disso, será criada uma pasta chamada "venv", com duas pastas dentro dela, "bin" e "lib". Precisaremos, agora, ativar a venv para que consigamos continuar a utilizar o projeto.

Na pasta "bin", há o arquivo "activate". No terminal, vamos fazer o caminho até chegar a esse arquivo. No macOS, utilizamos o comando source venv/bin/activate. Já no Windows, o comando, apesar de similar, é diferente: venv\Scripts\Activate.

Nota: Se quisermos desativá-lo futuramente, basta executar o comando deactivate

Agora precisamos instalar o Django. Faremos isso usando com a ajuda do Pip, um programa de gerenciamento de pacotes do Python.

Basta rodar o comando pip install django

# Servidor

Pip freeze

Agora que o Django está instalado, vamos executar uma boa prática em programação.

Existe uma forma de visualizar todas as dependências do projeto e os módulos que precisam ser instalados para que o projeto funcione. Para fazer isso, executamos o comando pip freeze no terminal. Ele informa a versão de Django, sqlparse e asgireft necessárias ao funcionamento do projeto. **Uma boa prática é criar um arquivo, chamado "requirements.txt", contendo todos os arquivos necessários**.

Fazemos isso executando o comando pip freeze > requirements.txt.

Sempre que instalarmos algo, como o Django e o Flask, por exemplo, precsamos executar o comando pip freeze > requirements.txt para manter o arquivo atualizado.

Carregando o Django

Agora, vamos carregar o Django na nossa aplicação. Antes, porém, vamos executar o comando django-admin help no terminal. Com isso, poderemos visualizar todos os comando que podemos executar na ferramenta.

Dentre eles, usaremos **startproject** para iniciar nosso projeto. Vamos executar o comando django-admin startproject setup .

Fazendo isso, será criada uma pasta chamada "setup", dentro de "alura-space", com as configurações do projeto.

Se não adicionássemos o . em django-admin startproject setup ., seria criada outra pasta "setup", dentro de "setup", com as configurações.

Subindo servidor

**Python manage.py runserver**

# Idioma e Timezone

As configurações relacionadas ao idioma da aplicação ficam em "setup > settings.py".

Nesse arquivo, encontramos todas as configurações do projeto, como dependências, templates e mais. Portanto, vamos usá-lo muito em nossas operações, porque precisaremos manipular essas configurações. Nas linhas 106 e 108 do código do arquivo,encontramos "LANGUAGE\_CODE" e "TIME\_ZONE", exatamente o que procurávamos.

Ao lado de "LANGUAGE\_CODE", encontraremos 'en-us', que representa o inglês dos Estados Unidos. Vamos substituir para 'pt-br'. Com isso, vamos alterar a linguagem para português brasileiro. Em "TIME\_ZONE", usaremos 'America/Sao Paulo'. Isso será o suficiente para ajustar nosso fuso horário.

# Variáveis de ambiente

Agora que o projeto está configurado com idioma e horário, precisamos versionar o projeto usando Git e Github.

Porém, não podemos enviar todas as partes do código para essas ferramentas, por questões de segurança. Se enviássemos o código por inteiro, receberíamos uma mensagem do Github, avisando que uma Django Secret Key está acessível no Github. A linha que contém a Secret Key, chave secreta, está em "setup > settings.py".

Todo projeto Django tem uma chave secreta. Essa key não deve ser disponibilizada para outras pessoas.

No terminal, vamos solicitar que o Pip instale uma nova dependência, com o comando pip install python-dotenv. **Depois da alteração, vamos atualizar o arquivo de Requirements com pip freeze > requirements.txt.**

Vamos criar uma novo arquivo na pasta "Django\_4", chamado ".env". Nele, vamos inserir a Secret Key, seguindo o modelo abaixo:

**SECRET\_KEY = sua secret key**

Agora, podemos remover a Secret Key do arquivo "settings.py"

Vamos carregar, agora, as variáveis de ambiente.

Na linha 13, onde importamos Path, vamos importar também o os, utilizando from dotenv import load\_dotenv.

Abaixo, criaremos a função load\_dotenv()Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Agora, vamos descer até a linha 26 do código, onde antes havia a Secret Key. Vamos passar a variável transformada em string

str(os.getenv('SECRET\_KEY')) no lugar da Secret Key.

Ex:

**Texto

Descrição gerada automaticamente**

Agora, quando enviarmos para o Github, ele apresentará a string str(os.getenv('SECRET\_KEY')) ao invés da Secret Key, que está armazenada somente no nosso computador. Isso acontece porque nós não enviamos o arquivo .env para nosso repositório.

# Git e Github

Enviando todo o código para o Github

Para isso, criaremos um novo repositório na página do nosso Github, clicando em "+ > New repository". Daremos o nome "alura\_space" a ele. Agora já temos o local para onde enviaremos o código.

Dica: Diversas pessoas que programam com Django têm isso como rotina. Por isso, já existe um padrão para fazer isso. Vamos acessar gitignore.io. O site nos perguntará em qual linguagem estamos codando. Portanto, vamos digitar "Django". Receberemos a definição de cada arquivo que não podemos enviar para o Github. Vamos apertar "Ctrl + A", para selecionar tudo, e "Ctrl + C", para copiar. De volta ao código, apertaremos "Ctrl + V". Com isso, copiaremos todas as informações para o nosso ".gitignore".

Vamos iniciar um repositório local com a ajuda do comando **git init**. Depois, vamos executar git add ., que copiará tudo o que precisamos.

Os arquivos que não serão enviados ao Github serão exibidos na cor cinza na barra de exibição lateral. Vamos realizar o commit com git commit -m "projeto alura space. Depois disso o commit será criado.

Vamos acessar o repositório que criamos no Github. Nele, na seção "...or create a new repository on the command line", copiaremos a linha que traz, além de git remote add origin, o link do repositório. No caso do instrutor, o comando era git remote add origin https://github.com/guilhermeonrails/alura\_space.git, mas o link será diferente para cada repositório.

Vamos copiar esse comando e executá-lo no terminal. Depois, executaremos git push origin master.

Tela de celular com aplicativo aberto

Descrição gerada automaticamente

# Para saber mais: portas do servidor

Geralmente temos um número vasto de aplicações e programas instalados na nossa máquina física. Muitas dessas ferramentas se conectam na internet e realizam comunicações com servidores distantes, trocando informações constantemente.

A partir do avanço dos computadores e da internet, houve a necessidade de organizar melhor o meio pelo qual determinadas aplicações contidas em cada máquina se comunicam.

Para essa tarefa, foi criado o **Transmission Control Protocol (TCP)** - **Protocolo de Controle de Transmissão**. Esse protocolo utiliza de artifícios chamados de **Portas de Comunicação** para gerenciar as trocas de informações de várias aplicações do sistema instaladas na mesma máquina física.

Aqui estão algumas das Portas de Comunicação mais conhecidas e suas atribuições:

| **Porta** | **Atribuição** |
| --- | --- |
| 80 | HTTP |
| 433 | HTTPS |
| 22 | SSH |
| 53 | DNS |
| 25 | SMTP |
| 465 | SMTPS |
| 20, 21 | FTP |
| 67, 68 | DHCP |

# App e projeto

O comando startapp diz respeito a um pedaço da aplicação com funcionalidades específicas. Já o comando startproject se relaciona à coleção de todas as configurações de uma determinada aplicação. **Um projeto pode abrigar, portanto, diversos apps**.

Quando falarmos em funcionalidades da nossa aplicação, estamos trabalhando com apps. Quando os criamos, eles podem ficar em lugares diferentes. Vamos criar um app chamado "galeria", porque seu objetivo será manter todo o código relacionado às imagens e fotos do nosso programa.

Para isso, usaremos o comando python manage.py startapp galeria. Quando executarmos, veremos que uma pasta chamada "galeria" será criada dentro da pasta do nosso projeto. Agora precisamos sinalizar que o app "galeria" faz parte do nosso projeto. Faremos isso acessando "setup > settings.py".

Vamos rolar a página até encontrar a seção "INSTALLED\_APPS" do código.

ISNTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'galeria',

Views e URLs

Criamos um app chamado "galeria", e uma função, no arquivo de view, que tem como respostas o que visualizamos no navegador.

Função abaixo:

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

O direcionamento está dentro de urlpatterns. Isso é viável, por enquanto, porque criamos apenas um app. Mas se tivermos três apps ou mais, por exemplo, não fará sentido fazer essa requisição dessa maneira, porque, assim, precisamos de uma linha de código para cada rota

Para solucionar isso, ao invés de usarmos o arquivo "urls.py" para listar as páginas de todos os apps, vamos criar uma maneira de isolar esse roteamento dentro da nossa galeria. Vamos criar um arquivo chamado "urls.py" dentro da pasta "galeria".

Nesse arquivo, vamos trazer o método import path e a função de views, importando o método index. Depois, criaremos uma lista para manter todos os endereços da nossa aplicação relacionados à galeria, com o comando urlpatterns, dentro do qual criaremos uma lista, abrindo chaves.

Dentro da lista, criaremos um path('', index):

Texto

Descrição gerada automaticamente

Agora, precisamos tornar o urlpatterns desse arquivo disponível no arquivo "urls.py" da pasta "setup".

Em django.urls, vamos importar, além de path, um método chamado include.

Dentro dos parênteses, vamos inserir o nome do aplicativo, "galeria", e urls:

Texto

Descrição gerada automaticamente

Templates

Nós não queremos desenvolver toda a nossa aplicação dentro de "galeria > view.py", no retorno de HttpResponse.

Queremos que haja uma página HTML e que ela seja renderizada. Para fazer isso, precisaremos dizer ao nosso projeto onde ficam suas páginas HTML. Vamos acessar "setup > settings.py" e rolar a página até a seção TEMPLATES, na linha 58 do código.

Essa parte do código abriga toda a parte visual da aplicação. Ao lado de 'DIRS', informaremos o local dos arquivos HTML. Dentro da pasta "alura-space", criaremos a pasta "templates", que será a pasta passada em 'DIRS'. Faremos isso usando os.path.join():

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

Como queremos mostrar algo através dessa nova pasta, criaremos uma nova aplicação dentro dela, chamada "index.html". Vamos inserir ! na primeira linha do código, para gerar a estrutura base. Dentro de <body>, adicionar um <h1> onde se lê "Nossa... deu certo mesmo":

<body>

<h1>Nossa... deu certo mesmo!</h1>

</body>

Acessando "migrations > views.py", vamos remover o HttpResponse, na linha 5, e o from django.shortcuts import render da linha 2. Vamos levar a propriedade render para o retorno, passando como parâmetros request, que será sempre o primeiro parâmetro nesse caso, e o arquivo que queremos exibir entre strings. No caso, é index.html:

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Carregando os templates

Para começar, vamos abrir nosso projeto Django. Para manter nosso código mais organizado e isolado, vamos criar, dentro da pasta "templates", a pasta "galeria". Dentro dela, vamos colocar o arquivo "index.html" que, até agora, estava na pasta "templates".

Se voltarmos ao navegador e atualizarmos a páginas, vamos nos deparar com uma página de erro, que informa que o template "index.html" não existe. Isso acontece porque não informamos sua existência ao arquivo "galeria > migrations> view.py".

from django.shortcuts import render

def index(request):

return render(request, 'galeria/index.html')

Agora vamos acessar o arquivo "index.html" do código disponibilizado na tarefa anterior. Com a ajuda de um "Ctrl + A", vamos selecionar todo o código.

Vamos copiá-lo, com "Ctrl + C", voltar ao arquivo "templates/galeria > index.html", selecionar todo o código com "Ctrl + A" e substiui-lo pelo código que acabamos de copiar. Vamos salvar a alteração e voltar ao navegador.

Na sequência, vamos importar esses estilos, para carregá-los no nosso navegador.

Arquivos estáticos

Agora vamos carregar os arquivos estáticos da aplicação. O caminho será similar ao dos arquivos HTML.

Na linha 61 de "setups > settings.py", informamos que há uma pasta onde manteremos todos os nossos códigos HTML. Faremos um processo semelhante, mas para indicar o caminho dos arquivos estáticos.

Vamos descer a página até chegarmos à linha 122, onde lemos STATIC\_URL = 'static/'. Vamos informar a pasta que receberá esses arquivos, escrevendo, na linha 124, STATICFILES\_DIRS, passando o caminho os.path.join, com os parâmetros BASE\_DIR, 'setup/static'.

Precisamos passar, também, o caminho absoluto para que o diretório consiga pegarr os arquivos estáticos. Para isso, vamos inserir a "raiz" dos caminhos com STATIC\_ROOT, com o caminho `os.path.join(BASE\_DIR, 'static'):

**STATIC\_URL = 'static/'**

**STATICFILES\_DIRS = [**

**os.path.join(BASE\_DIR, 'setup/static')**

**]**

**STATIC\_ROOT = os.path.join(BASE\_DIR, 'static'**

Na pasta "setup", criaremos uma pasta chamada "static". Agora criamos o diretório que servirá para a manipulação dos arquivos estáticos.

Nosso projeto já tem HTML e CSS. Vamos copiar, por isso, os arquivos das pastas "assets" e "styles" e levá-los para a pasta "setup > static". Vamos abrir o terminal, executar "Ctrl + C", para fazê-lo funcionar, e carregar a venv com o comando python manage.py runserver.

De volta ao navegador, se atualizarmos a página, nada mudará. Apesar disso, já temos os arquivos estáticos dentro da aplicação. Existe um comando para solicitar que o Django manipule os arquivos estáticos da aplicação, para que possamos visualizá-los.

Trata-se do comando collectstatic, que pode ser encontrado acessando a lista de comandos via CLI, passando python manage.py help. De volta ao terminal, vamos rodar o comando python manage.py collectstatic:

**python manage.py collectstatic**

Depois de executarmos o comando, percebemos a criação de uma pasta chamada "static", relacionada ao STATIC\_ROOT que criamos. Agora, já podemos usar os arquivos estáticos no Django. Se rodarmos o servidor outra vez, com python manage.py runserver, não vamos notar alterações no projeto.

Para realizar as alterações, vamos acessar "templates/galeria > index.html". Vamos adicionar o código {% load static %}, que solicita o carregamento dos arquivos estáticos, à primeira linha do arquivo.

Vamos carregar, inicialmente, o arquivo "style.css", o caminho para encontrá-lo é "static > styles > style.css". Na linha 13 do código, vamos adicionar {% static após o href e %} após o fechamento das aspas, o que identifica um arquivo estático a ser subido.

Também vamos colocar styles/style.css entre aspas simples e levar as aspas duplas para depois do fechamento das chaves:

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

Atenção: Quando usamos código Python junto ao HTML, chamamos de "embedado". Porque fizemos o embedding do código Python ao HTML.

Carregando as imagens