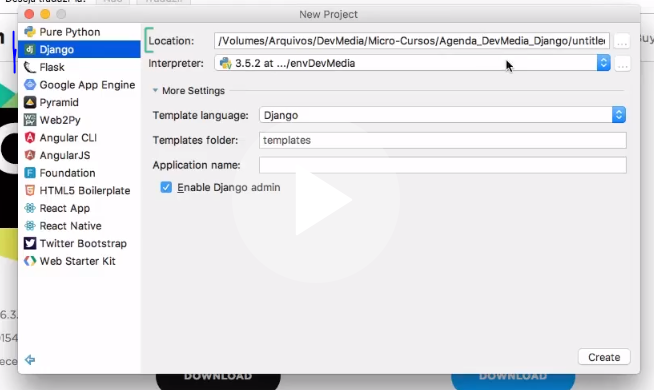
**Django | Python**

O Django é um framework em Python que serve para fazermos aplicações Web com o Python.

Vamos ver como o Django funciona na prática abaixo:

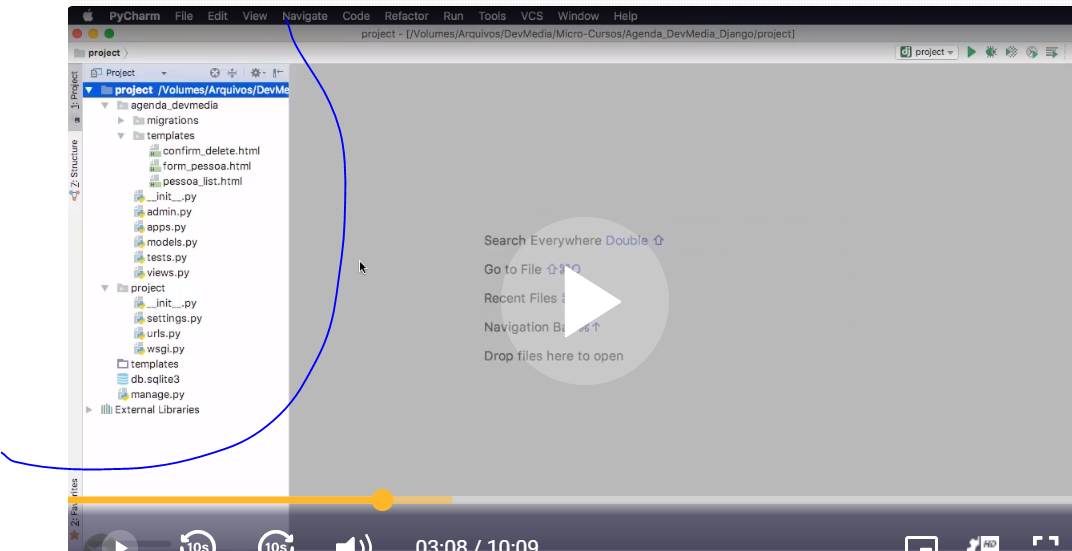
Precisamos baixar o Python e o framework Django para utiliza-lo.

Para fazer um projeto Django, vamos no Pycharm e selecionamos um novo projeto Django:



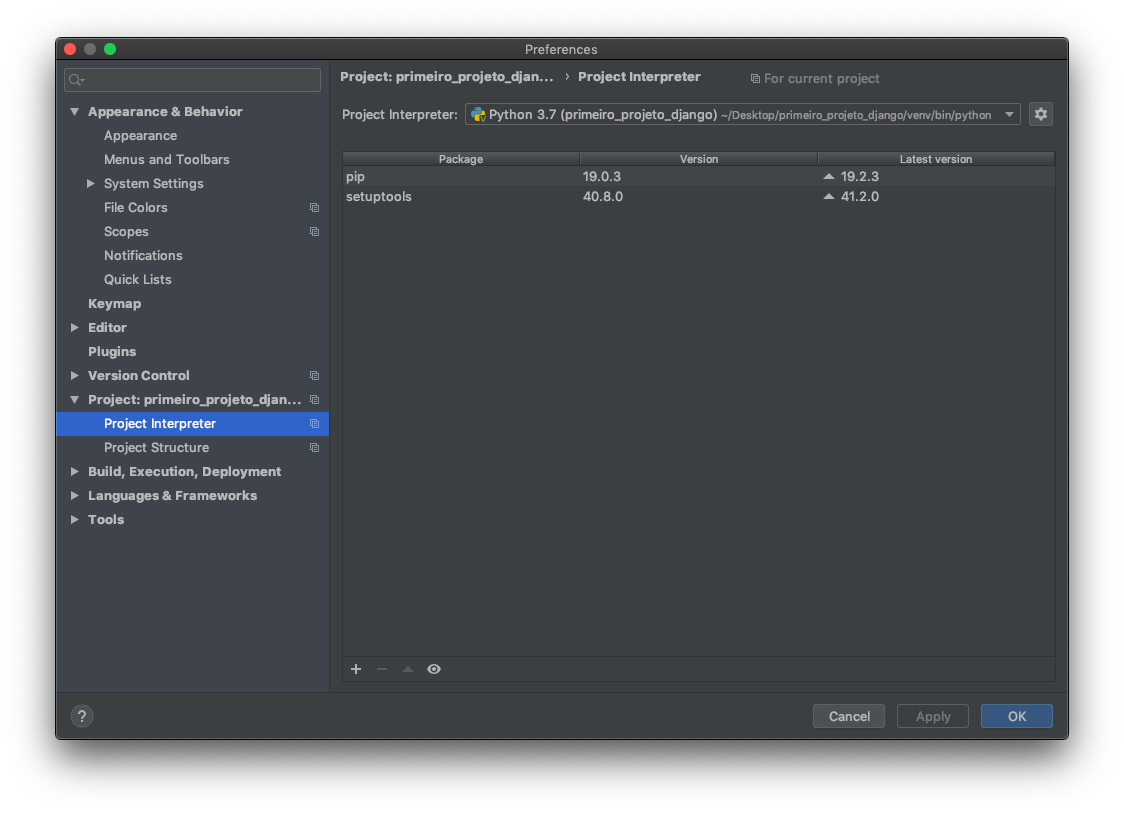
Acima também escolhemos a template language com o Django, a template folder que é onde ficarão guardados nossos templates e o nome da aplicação.

Com isso o PyCharm se encarregará de baixar o Django na versão mais recente e instalar na nossa pasta.

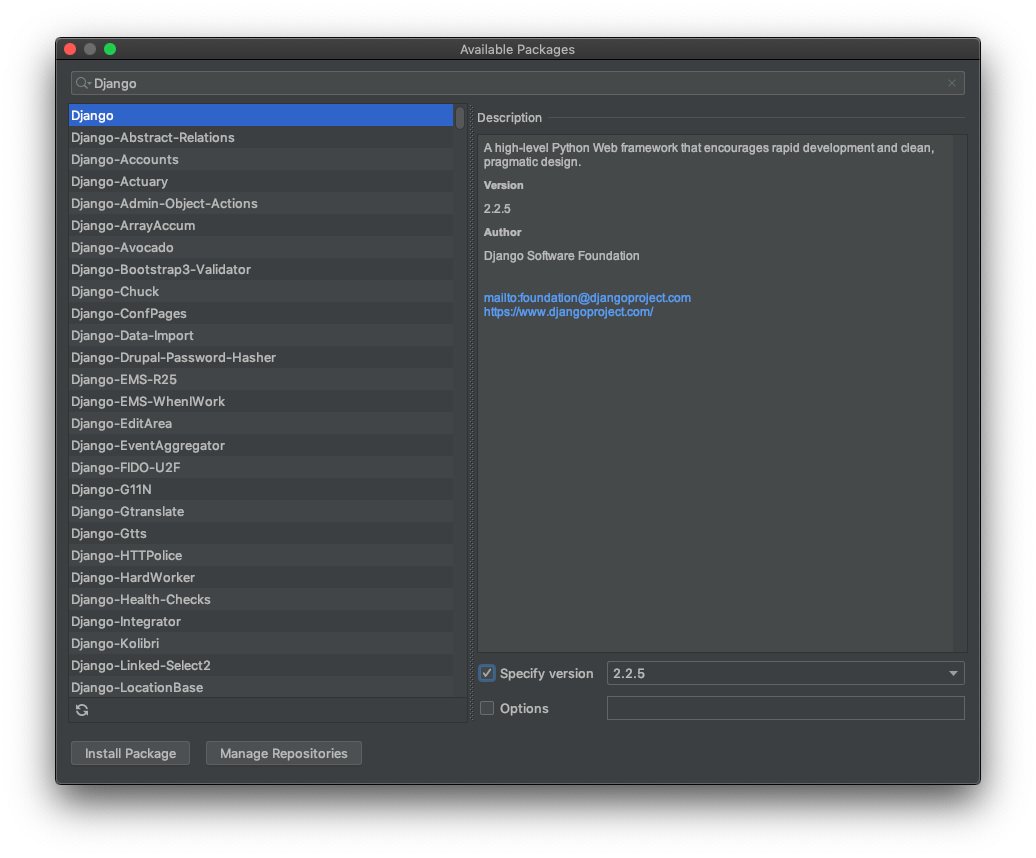


Podemos ver que nas pastas tem os arquivos de templates da minha aplicação Web, como os arquivos de configurações. De URLs do projeto e também de Models, Views e tests.

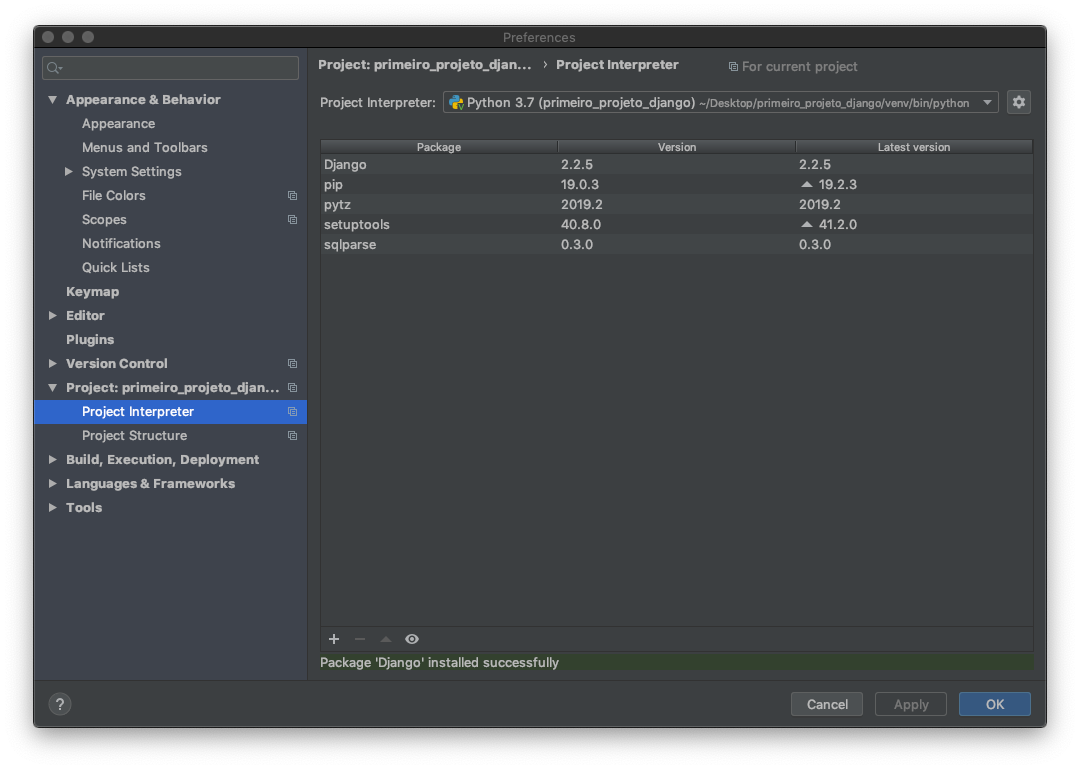
Com o projeto do PyCharm criado, o primeiro passo é instalar o Django em nossa virtualenv. Para isso, o PyCharm possui uma interface gráfica que auxilia todo este processo, localizada em File > Settings > Project Interpreter (se você está utilizando o Windows ou Linux) ou PyCharm > Preferences (se você está utilizando o macOS).



É nesta janela que poderemos adicionar os pacotes à nossa virtualenv. Para isso, clicamos no botão “+” localizado no canto inferior esquerdo da janela, que exibirá uma tela para buscarmos pelo pacote desejado:



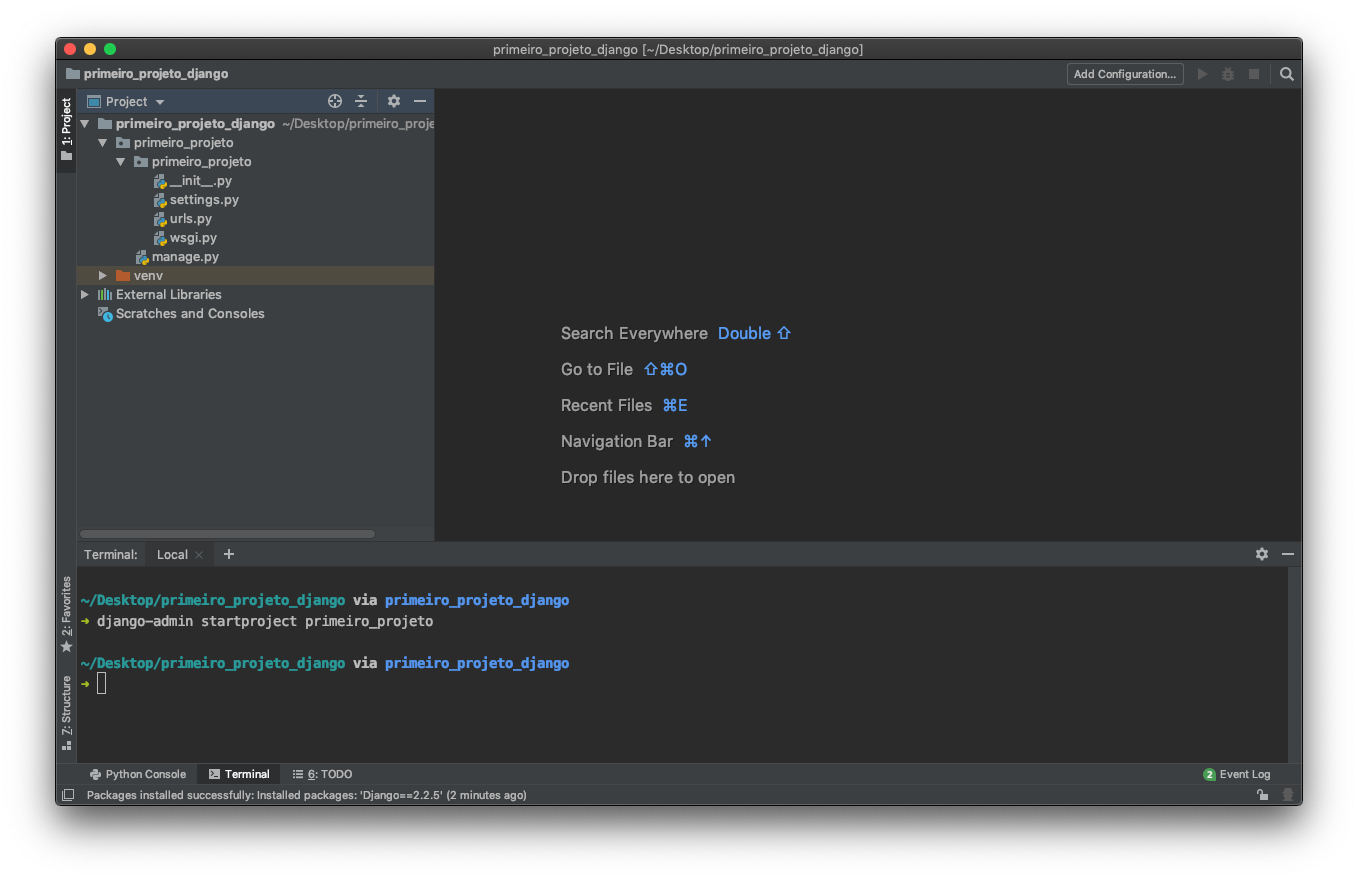
Nesta janela, buscamos pelo pacote “Django” e clicamos em “Install Package”. Após isso, o PyCharm (utilizando o PIP) vai baixar o pacote selecionado e instalar na virtualenv do projeto. Ao finalizar este processo, podemos fechar a janela e notaremos que o pacote foi instalado com sucesso:



Agora, voltando ao PyCharm, precisamos criar o projeto utilizando o Django. Para isso, no terminal integrado da IDE, digitamos o seguinte comando:

**django-admin startproject primeiro\_projeto**

O comando acima irá criar um projeto Django chamado “primeiro\_projeto” com a seguinte estrutura de pastas:



Basicamente, há quatro arquivos no projeto criado, onde a função de cada um deles pode ser vista abaixo:

* **settings.py:** Arquivo para armazenar as configurações gerais do projeto, como os pacotes a serem utilizados, configurações de bancos de dados, localização de arquivos estáticos, etc.
* **urls.py:** Arquivo para armazenar as rotas que serão utilizadas no projeto. **Este arquivo armazenará as rotas do projeto em geral, é recomendável que cada aplicação do projeto possua um arquivo de rotas específico**… Mas não se preocupe, falaremos sobre apps no próximo artigo :)
* **wsgi.py:** Interface simples e universal para troca de informações entre servidores web e aplicações criadas com Python.
* **manage.py**: Arquivo responsável por gerenciar o projeto como um todo. É, basicamente, um CLI para projetos Django.

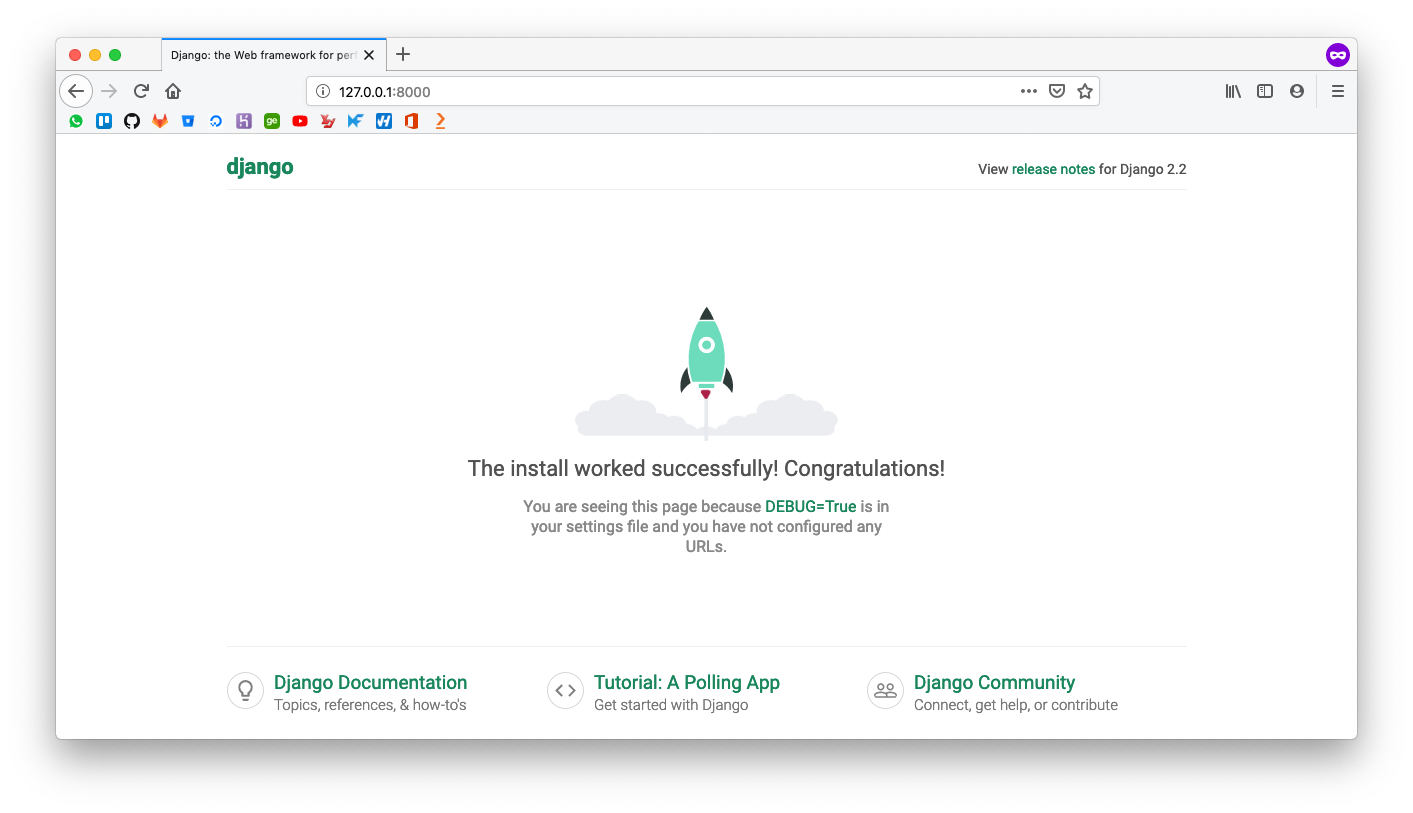
**Executando servidor Django**

Com o projeto criado, podemos executar seu servidor de desenvolvimento para verificar se tudo está conforme planejado. Para isso, ainda no terminal do PyCharm, utilizamos o seguinte comando:

**python manage.py runserver**

Ao executar o comando acima, o servidor do Django será inicializado e poderemos acessá-lo através da rota **http://127.0.0.1:8000**, conforme é retornado no terminal:

Ao executar a rota citada acima em um navegador, teremos a seguinte página como resposta:



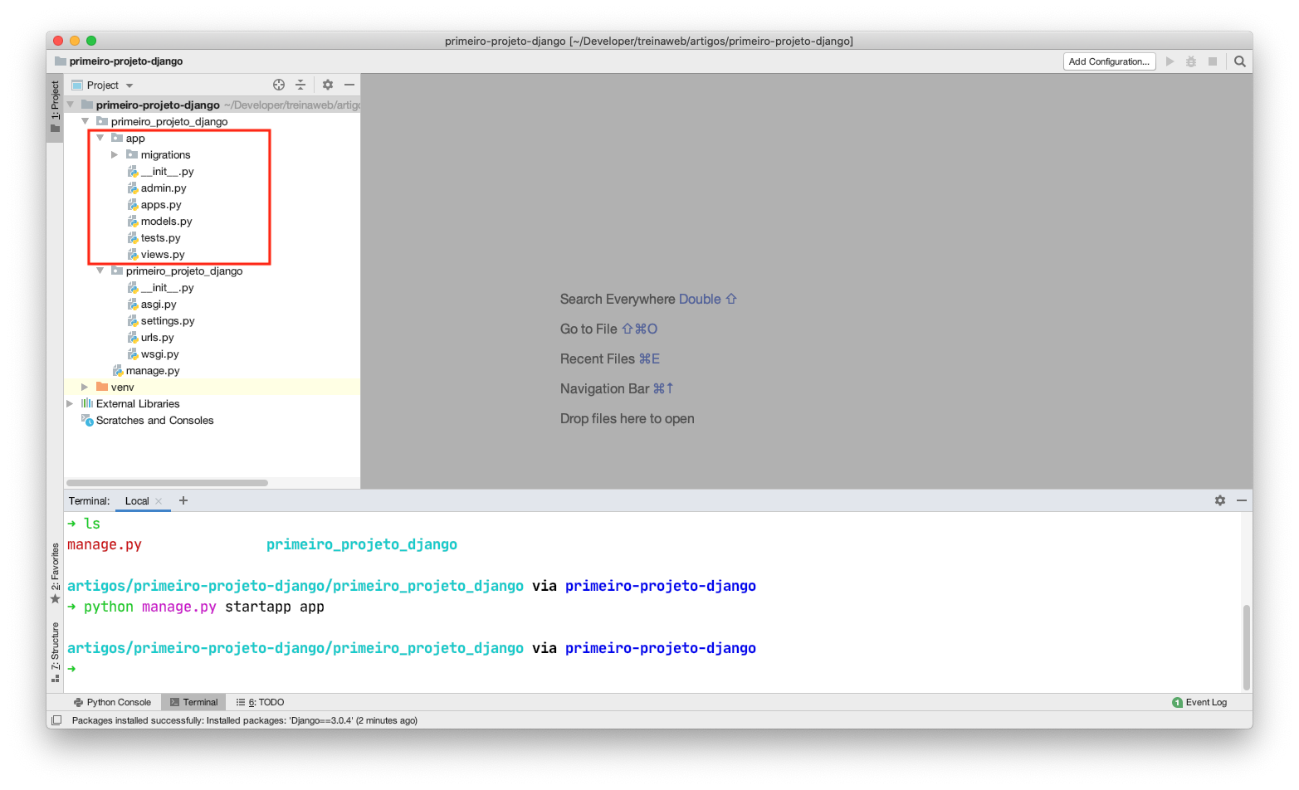
Esta (linda) página indica que a instalação foi feita corretamente e que estamos prontos para desenvolver nossa primeira aplicação com o Django o/

Com o projeto criado, o próximo passo é criar nossa aplicação. Basicamente, o Django é dividido em diversas apps em um mesmo projeto. Isso facilita o reaproveitamento de código e organização do projeto como um todo.

Para criar uma app no Django, utilizamos o seguinte comando:

**python manage.py startapp nome\_da\_app**

Ao executar o comando acima (alterando o nome\_da\_app para o nome da sua aplicação), uma série de arquivos é criada:



Estes arquivos são:

* Diretório migrations: Diretório responsável por armazenar os arquivos de migração para o banco de dados de uma aplicação Django.
* admin.py: Arquivo responsável por definir os *models* a serem utilizados no [módulo administrativo do Django](https://www.treinaweb.com.br/blog/criando-uma-aplicacao-administrativa-com-django-admin/).
* apps.py: Arquivo responsável pela configuração da app do projeto Django.
* **models.py**: Arquivo responsável por definir os modelos da aplicação. Basicamente, um modelo é a representação das tabelas a serem criadas no banco de dados.
* tests.py: Arquivo responsável por definir as regras de testes da aplicação.
* **views.py**: Arquivo responsável por definir as regras de negócio do app.

Ao criar a aplicação, precisamos registrá-la no arquivo settings.py do projeto. Isso fará com que o projeto Django saiba que essa app faz parte do projeto e que ele pode gerenciá-la. Para fazer isso, no arquivo settings.py, adicionamos o nome da app no array INSTALLED\_APPS, como podemos ver abaixo:

Feito isso, o projeto Django já sabe que a aplicação app faz parte do projeto e já estamos prontos para iniciar seu desenvolvimento,

## Criar Migrações

Após determinar quais serão as entidades que utilizaremos em nosso projeto, precisamos criar as tabelas que representam estas entidades no banco de dados. Para isso, utilizamos o seguinte comando:

Copiar

**python mananage.py makemigrations**

O comando acima irá **obter a estrutura das classes definidas no arquivo models.py** e criará os arquivos de migração de cada classe.

## Executar Migrações

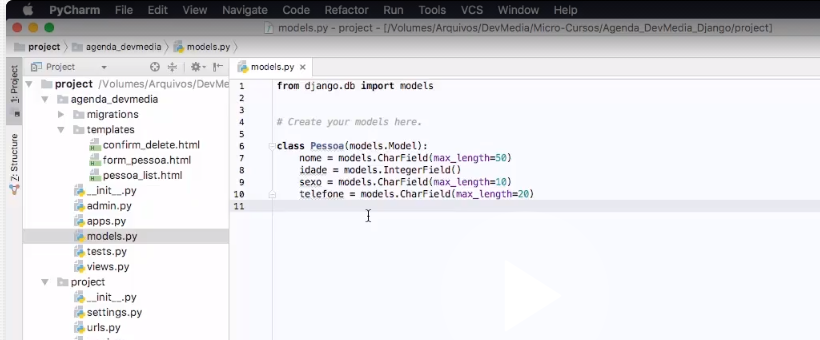
Após criar os arquivos que definem a estrutura de cada entidade no banco de dados, precisamos executar estas migrações. Para isso, utilizamos o seguinte comando:

Copiar

**python manage.py migrate**

Com isso, todas as classes definidas no arquivo models.py serão transformadas em tabelas no banco de dados.

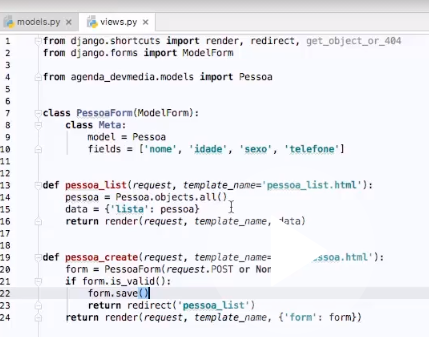
**Como o Django é baseado na Arquitetura MTV, Model, Template e View, temos um arquivo chamado models.py que podemos criar os Models, ou seja, as classes de nossa aplicação em Django. É neste arquivo que vamos colocar os Models da nossa aplicação:**



from django.db import models  
  
#Create your models here.  
  
class Pessoa(models.Model):  
 nome = models.CharField(max\_length=50)  
 idade = models.IntegerField()  
 sexo = models.CharField(max\_length=10)  
 telefone = models.CharField(max\_length=20)

Veja que usamos para os tipos dos campos “CharField e IntegerField” e para usar estes tipos importamos de Django.db o “models”. **Também repare que as nossas classes herdam um Pai do tipo “models.Model”.** Todas as nossas classes devem seguir este padrão.

**Após criar nossos Models, nós vamos criar nossas Views:**

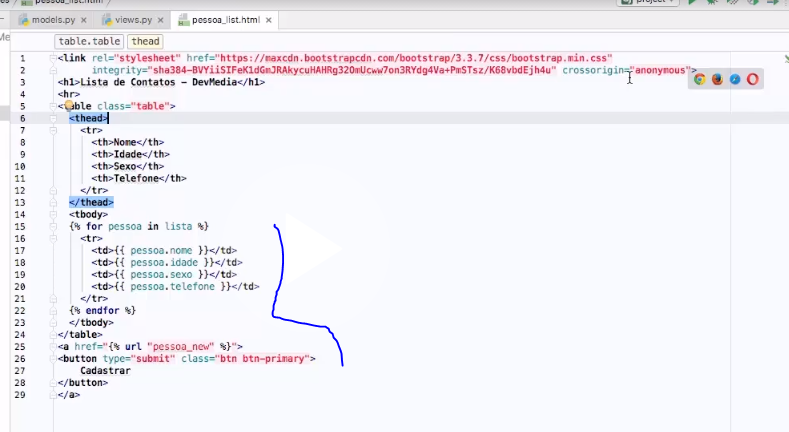


Então, conforme acima, após criar nossa View, primeiro vamos criar uma classe PessoaForm que será um formulário que usaremos mais tarde relacionando o Model que criamos Pessoa. Note que o padrão tem que ter em “Fields” os mesmos campos que definimos no Model de Pessoa.

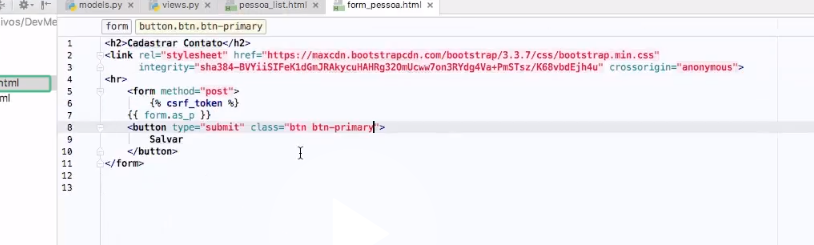
Depois disso, dentro da View, criamos duas funções:

* pessoa-list: que vai listar todas as pessoas cadastradas no banco de dados. Essa função vai passar um request e o nome do template. A gente recupera depois todas as pessoas através do método “Pessoa.objects.all()” e coloca na variável “pessoa”. Aí depois criamos uma variável “data” onde definimos que a variável lista é do tipo pessoa, ou seja, de pessoa. E por fim, vamos informar através do render que queremos passar para o nosso template\_name que setamos no argumento da função todas as pessoas que estão armazenadas na variável “data”.
* pessoa-create: Novamente vamos passar um request e um template\_name. Depois atribuímos a varivel form o PessoaForm que criamos no inicio da View passando como requisição do tipo POST. Depois vamos verificar se o formulário é valido e se for valido ele vai salvar os dados e vai por fim redirecionar para nossa URL pessoa\_list para que possa verificar aquela pessoa nova criada. E vai terminar tudo fazendo um request no nosso template\_name que passamos como argumento com a nossa variável form.

**Após criar nossos Models (modelos e classes) e Views (que nessa caso são os controllers da nossa aplicação), vou criar os TEMPLATES da nossa aplicação:**



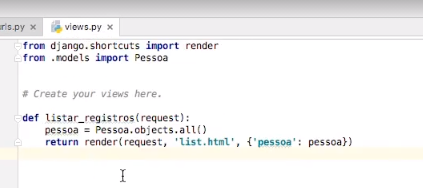
Neste template estou criando uma tabela e coloco no corpo da tabela um for com todos os valores de pessoa da nossa lista que definimos no data da nossa View, colocando os campos de cada pessoa neste corpo. Colocamos um linkezinho “url “pessoa\_new” que vai direcionar para o template de Formulario.



**O nosso template de formulário tem uma tag form do tipo POST. E usando o código “{{ form.as\_p }}” o Django já pega todos os campos daquele Form que fizemos no View e coloca neste formulário aqui no Template. Com isso já criamos nosso formulário e estamos prontos para enviar os dados para o banco de dados.**

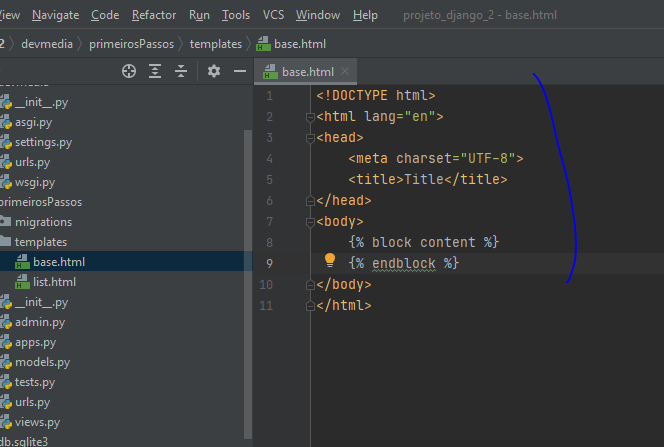
Em resumo, vamos ver como criar nossas views e fazer nossas comunicações com nossos templates:

Após direcionar uma URL utilizando rotas para um view eu posso colocar funções e instruções como abaixo:

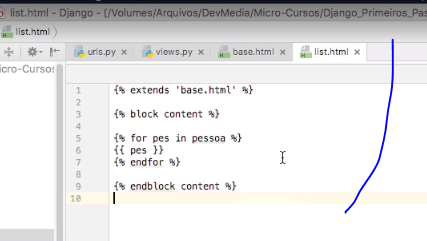


* **o comando acima “Pessoa.object.all()” serve para pegar todos os dados do banco de dados da Tabela/Objeto Pessoa.**
* O comando “return render(request, ‘list-html’, {‘pessoa’: pessoa}) serve para retornar **na** template list.html os dados que peguei anteriormente do banco de dados.

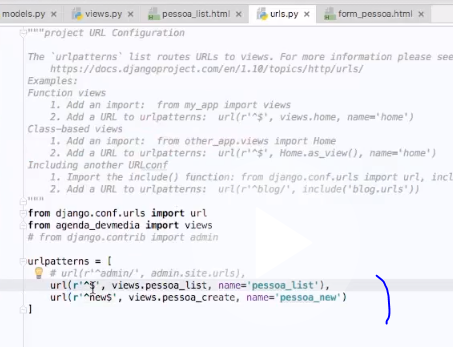
**Vale lembrar também em relação aos templates que com o Django nós definimos um template HTML como base e nele definimos com as tags o que vai ser igual para todas as páginas e o que vai mudar com o padrão de blocos de código em colchetes:**



**E aí dentro deste bloco será renderizado o código de outro template html usando esse padrão de colchetes, nesse exemplo o template HTML list que renderizará a lista de clientes sem precisar usar HTML e só usando o padrão de colchetes que extende a base.html:**



**Por fim, eu defino no arquivo URLS as minhas URLs de cada template:**



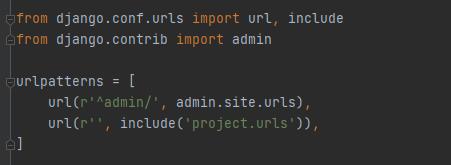
Sendo a nossa URL padrão a primeira linha, que é a de pessoa\_list e a URL de pessoa\_create onde posso chama-la nos views de pessoa\_new.

**Para gente entender este padrão para definir a rota no arquivo de rotas, segue abaixo:**

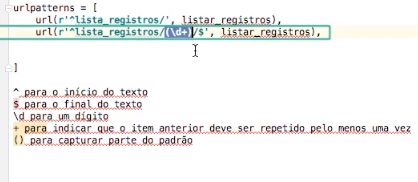


* Colocamos dentro de “r’^ ‘” a URL que queremos. Se fosse uma URL “admin/”, colocaríamos “r’^admin/’”;
* O segundo parâmetro significa a view que vai ter passar/executar quando estiver nessa URL que passei anteriormente. **Ou seja, quando eu entrar em uma URL vai entrar numa view que vai fazer os tratamentos necessários, como chamar um template, uma** função, etc...
* O último parâmetro é o nome dessa rota. **Então lá nos links eu utilizo uma rota com o nome que definir aqui.**

**Vale lembrar que temos no início do projeto, como macro um arquivo para configurações das URLs. Para que estas rotas macros possam apontar para as rotas da minha aplicação e não globalmente do projeto, eu crio dentro da aplicação um arquivo urls também e faço algo parecido com o abaixo no arquivo urls global:**



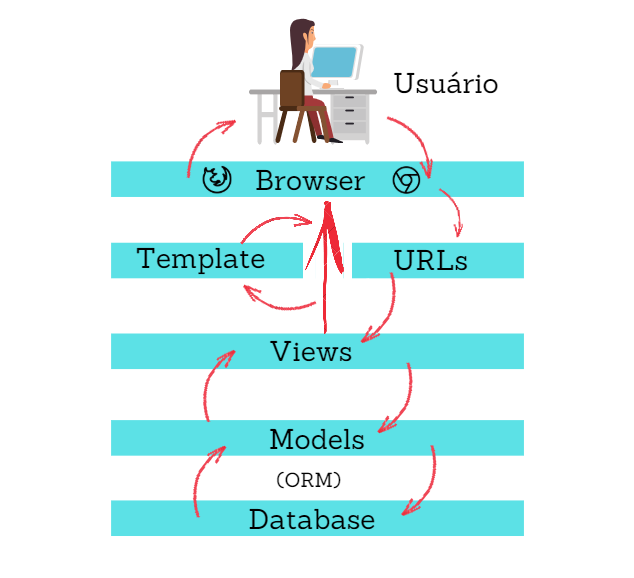
Estas urls utilizam padrão regex, segue um exemplinho de uso:



As URLs são extremamente importantes dentro da nossa aplicação, com elas, definimos que, para cada rota, determinada view será chamada. No Django, utilizamos expressões regulares para a criação das rotas

Então eu tenho esses três passos:

* Criar os models da minha aplicação:
* No view colocar todas as configurações da minha aplicação, utilizando a interligação do arquivo de Models que defini e os arquivos de templates que foi definir;
* Nos templates eu utilizo as variáveis e definições que fiz no arquivo de View.



**Como visto na imagem, quando o usuário faz uma requisição pelo browser, utilizando uma rota, é executado um método das Views, que utiliza os Models para acessar o banco de dados e retornar as informações. Estas informações são renderizadas pela camada de Template e, finalmente, é renderizado para o usuário pelo navegador.**

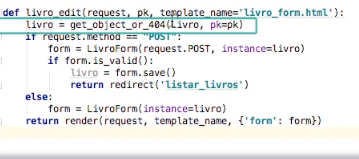
Resumidamente, a equipe de desenvolvimento do framework entende que a camada view descreve quais dados serão apresentados ao usuário, não a forma (aparência) que eles serão exibidos. Portanto, no padrão MTV, uma view é uma função que retorna algo para uma solicitação, porque ela define apenas quais dados serão apresentados, não como serão mostrados.

Além disso, é sensato separar o conteúdo da apresentação (por questões de organização e padronização do código). É aí que entra camada template. Com os dados retornados pela view, a template fica responsável por definir a forma que esses dados serão apresentados, normalmente em páginas HTML.

Porém, onde o controller se encaixa? No Django, os desenvolvedores do framework entendem que esta camada é a própria estrutura do projeto. É este mecanismo que envia uma solicitação para a view adequada de acordo com a configuração de rotas do Django. E isso faz sentido! Se pararmos para pensar, o controlador da aplicação é o próprio conjunto de bibliotecas que compõe o projeto. A view é mais adequada a retornar os dados a serem visualizados, que serão exibidos ao usuário por meio de templates.

**Podemos definir nas views ainda um método legal que serve para editar dados do banco de dados usando uns recursos do Django como abaixo**:

**Para editar um livro, precisamos recuperar, através da chave primária enviada através da URL, determinado livro cadastrado no banco de dados.** **Após isso, renderizamos um formulário com a instância do livro encontrado e, caso o formulário for válido, salvamos as alterações no banco de dados, caso o formulário não for válido, retornamos, novamente, para o formulário de edição** :

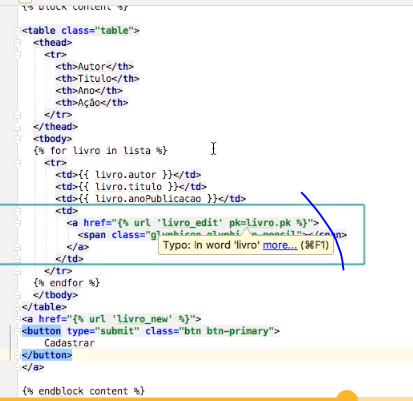


**No exemplo acima eu to pegando uma PK no banco de dados para pegar a PK do elemento que eu quero editar**. **Eu pego essa PK eu utilizo no método get\_object\_or\_404 que serve para pegar um elemento no banco de dados por sua PK ou se não achar retornar um erro 404.**

**Para pegar essa PK eu pego como parâmetro na URL**, **por isso coloco a configuração abaixo nas rotas de URL para pegar a PK e depois que pegar a PK pela URL chamar essa função de view utilizando essa PK que peguei pela URL**:



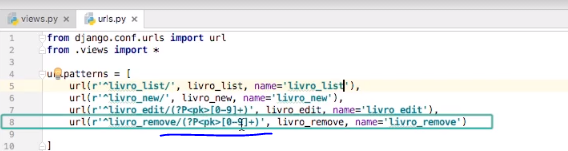
**Ai para ir pra esta view pelo template eu coloco no href de um botão que aparecerá na listagem de elementos, aproveitando-se do for:**



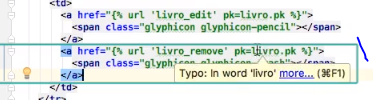
**Podemos também ter um método que retorna um objeto. Nesse caso utilizamos uma função do Django dentro das views .get que pega um dado no banco de dados com uma PK e aí com isso nós conseguimos deletar este dado no banco de dados**. Por consequência posso adicionar essa view para ser chamada em um botãozinho do for da lista de registros no template e quando for clicado nesse botãozinho e chamar essa view já vai excluir o registro e sumir do template:



Só que da mesma forma que o editar registros, eu pego essa PK pela URL também lá no arquivo de rotas:



No Template:



**Como fazemos para ter um relacionamento 1:N com o Django?**

Agora que já temos os CRUDs de produtos e marcas, precisamos realizar o relacionamento entre essas entidades a fim de definir que UM produto pertence a UMA marca e UMA marca pode possuir VÁRIOS produtos. **Para isso, no arquivo models.py do nosso projeto precisamos adicionar um atributo marca à entidade produto como podemos ver abaixo**:

class Produto(models.Model):

descricao = models.CharField(max\_length=50, null=False)

preco = models.FloatField(null=False)

**marca = models.ForeignKey(Marca)**

…

**Linha 4: Utilizando o método ForeignKey e passando a classe Marca como parâmetro, adicionamos um atributo marca na entidade Produto para que o relacionamento entre estas duas tabelas seja feita. Assim, um produto irá possuir apenas uma marca.**

O próximo passo é adicionar no array fields da classe ProdutoForm, localizado no arquivo views.py, o atributo marca como definimos em nosso model como podemos ver abaixo:

class ProdutoForm(ModelForm):

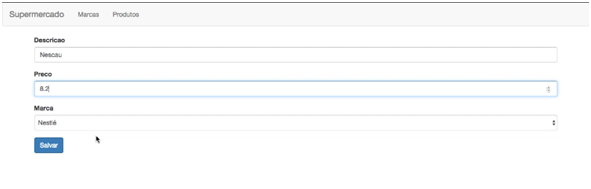
class Meta:

model = Produto

fields = ['descricao', 'preco', 'marca']

**Linha 4:**Adicionamos o atributo **‘**para indicar que um produto, agora possui além da descrição e preço, uma marca.

**Feito isso, o relacionamento entre produtos e marcas está concluído e sempre que um produto for cadastrado, deve-se selecionar uma marca como podemos ver na Figura 1**:

**Figura 1.** Cadastro de produto com relacionamento entre marcas

**Como fazemos para ter um relacionamento N:N com o Django?**

Até aqui possuímos as entidades Produto e Categoria, porém sem qualquer tipo de relacionamento entre elas. Sendo assim, precisamos definir a relação entre os models de forma que um produto pode pertencer a N categorias e uma categoria podem possuir N produtos. **Para isso, no arquivo models.py do projeto iremos adicionar um campo categoria dentro do model Produto como podemos ver abaixo**:

class Produto(models.Model):

descricao = models.CharField(max\_length=50, null=False)

preco = models.FloatField(null=False)

marca = models.ForeignKey(Marca)

**categoria = models.ManyToManyField(Categoria)**

**Linha 5**: Definimos que um produto possui várias categorias relacionadas a ele, usando para isso o método ManyToMany.

Agora precisamos adicionar o atributo categoria ao array fields da classe ProdutoForm, localizado no arquivo views.py, como podemos ver abaixo:

class ProdutoForm(ModelForm):

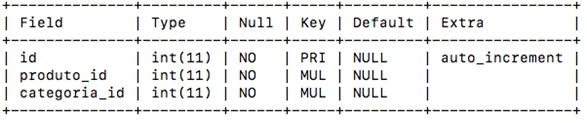
class Meta:

model = Produto

fields = ['descricao', 'preco', 'marca', 'categoria']

**Linha 4:**Adicionamos o atributo categoria para indicar que um produto possui uma ou mais categorias relacionadas a ele.

**Feito isso, será criada uma tabela no banco de dados responsável por armazenar os IDs dos produtos e categorias relacionadas como podemos ver nas Figuras 1 e 2:**

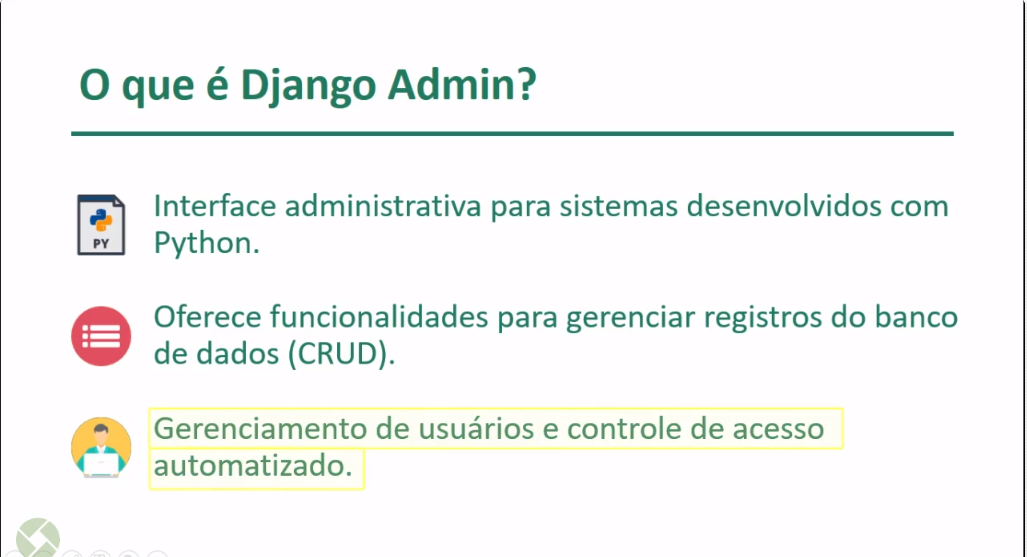
**Figura 1**. Tabela que armazena o relacionamento N:N**Figura 2**. Definição da tabela que armazena o relacionamento N:N

Agora, sempre que um produto for cadastrado no banco de dados, uma lista de categorias que podem ser relacionadas a ele será exibida, como podemos ver na **Figura 3**:

**Figura 3**. Cadastro de produtos com relação entre categorias

**Veremos agora sobre autenticação nas nossas aplicações utilizando Django:**

Com o Django Admin, podemos, facilmente, criar uma área administrativa para gerenciar todos os nossos modelos de dados e os usuários cadastrados no site. Podemos, também, realizar todo o controle de acesso dos usuários sem programar uma única linha de código.



**Com o Django Admin só basta a gente criar nosso model e sem desenvolvimento de código podemos criar nosso sistema e também auxilia no gerenciamento automatizado de usuários com controle de acesso. O Django Admin também tem um painel administrativo bastante completo para nosso sistema.**

**Usando o Django Admin temos que definir nossos models da mesma forma que definíamos sem o Django Admin, definindo nossos models dentro do arquivo models.py Não se esqueça de realizar o migrate do model no banco de dados.**

**Só que após definir esses models em models.py, nós registramos este model no arquivo “admin.py” para que o Django Admin saiba o que será manipulado, conforme abaixo:**

1 from django.contrib import admin

2 from .models import \*

3 # Register your models here.

4 **admin.site.register(Postagem)**

Linha 2 - Importando o arquivo .models da aplicação;

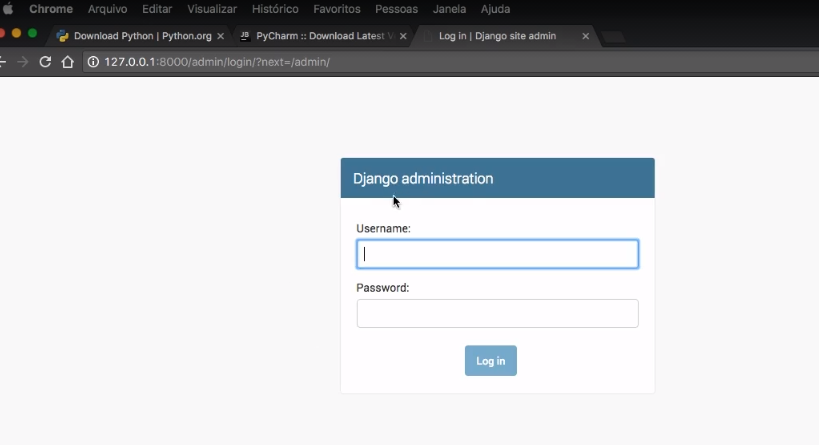
**Linha 4 - Registrando o modelo de Postagem no Django Admin.**

**Agora, precisamos criar o super usuário para conseguir acessar o módulo de administração com todos os privilégios, para que a gente consiga entrar na interface como superusuario:**

**python manage.py createsuperuser**

**Após executar esse comando, será pedido no terminal algumas informações como nome do superusuário, senha, email.**

**Depois que eu fizer isso, eu rodo o comando runserver para rodar a aplicação Django e para acessar a interface de admin do Django vamos na url localhost/admin e aparecerá a tela abaixo:**



Para que nosso Django Admin fique mais completo, precisamos definir duas configurações. A primeira é a exibição do nome do nosso model no plural. A segunda, é a exibição do título da postagem na listagem dos posts no Django Admin. Essas configurações podem ser vistas a seguir:

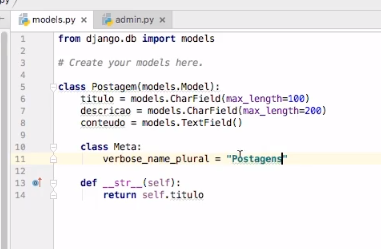
10 class Meta:

11 verbose\_name\_plural = "Postagens"

12

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

14 return self.titulo



**Lembrando que o Django Admin consigo gerenciar autenticação e permissões como, por exemplo, criação edição e exclusão de novos objetos do nosso model**. **Agora a parte de URLs, Template, View continuamos fazendo da mesma forma que aprendemos antes através dos códigos.**

Também podemos fazer uma lógica para criar nosso próprio programa de autenticação utilizando o código com Djando nas views. **Lembrando que o Django trabalha com dois tipos de usuários: SuperUsuário (root), Staff e Usuario comum.**

**O código abaixo é um exemplo de View contendo alguns métodos para um sistema de logins e registros...**

**from** django.conf **import** settings

**from** django.contrib **import** messages

**from** django.contrib.auth **import** authenticate, login, logout

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

**from** django.contrib.auth.models **import** User

**from** django.http **import** HttpResponse

**from** django.http **import** HttpResponseRedirect

**from** django.shortcuts **import** render, redirect

# Create your views here.

@login\_required

**def** registrar\_usuario(request, template\_name="registrar.html"):

user = request.user

**if** user.is\_staff:

**if** request.method == "POST":

username = request.POST['username']

email = request.POST['email']

password = request.POST['password']

tipo = request.POST['tipo\_usuario']

**if** tipo == "administrador":

user = User.objects.create\_user(username, email, password)

user.is\_staff = True

user.save()

**else**:

user = User.objects.create\_user(username, email, password)

**return** redirect('/listar\_usuario/')

**else**:

messages.error(request, 'Permissão negada.')

**return** redirect('/listar\_usuario/')

**return** render(request, template\_name, {})

@login\_required

**def** listar\_usuario(request, template\_name="listar.html"):

usuarios = User.objects.all()

usuario = {'lista': usuarios}

**return** render(request, template\_name, usuario)

**def** logar(request, template\_name="login.html"):

next = request.GET.get('next', '/listar\_usuario/')

//PARA PEGAR A PROXIMA PAGINA, SE NÃO TEM COMO, IR PARA A PAGINA PADRÃO QUE DEFINI COMO ‘LISTAR\_USUARIO’

**if** request.method == "POST":

username = request.POST['username']

password = request.POST['password']

user = authenticate(username=username, password=password)

**if** user **is** **not** None:

login(request, user)

**return** HttpResponseRedirect(next)

//VAI AUTENTICAR SE O USUARIO EXISTE NA LISTA DE OBJECTS USER, SE SIM, ELE VAI REALIZAR O LOGIN E CHAMAR O MÉTODO NA VARIÁVEL “NEXT” PARA REDIRECIONAR PARA UMA PÁGINA.

**else**:

messages.error(request, 'Usuário ou senha incorretos.')

**return** HttpResponseRedirect(settings.LOGIN\_URL)

//SE O LOGIN NÃO DER CERTO, EU REDIRECIONO PARA A URL PADRÃO DE LOGIN QUE DEFINI NO ARQUIVO EM SETTINGS.PY

**return** render(request, template\_name, {'redirect\_to': next})

@login\_required

**def** remover\_usuario(request, pk, template\_name='delete.html'):

user = request.user

**if** user.has\_perm('user.delete\_user'):

VERIFICAR SE O USUARIO LOGADO TEM PERMISSÃO PARA EXCLUIR OUTRO USUARIO

**try**:

usuario = User.objects.get(pk = pk)

VERIFICAR SE O USUARIO QUE EU QUERO EXCLUIR EXISTE NA LISTA DE OBJECTS USER

**if** request.method == "POST":

usuario.delete()

**return** redirect('listar\_usuario')

**except**:

messages.error(request, 'Usuário não encontrado.')

**return** redirect('/listar\_usuario/')

**else**:

messages.error(request, 'Permissão negada.')

**return** redirect('/listar\_usuario/')

**return** render(request, template\_name, {'usuario': usuario})

**def** deslogar(request):

logout(request)

**return** HttpResponseRedirect(settings.LOGIN\_URL)

FAZER O LOGOUT DO USUARIO LOGADO NO REQUEST E REDIRECIONAR PARA A TELA DE LOGIN PADRÃO QUE DEFINI NO ARQUIVO SETTINGS.PY

**E colocamos no template HTML de login no sistema o seguinte código, onde passamos no action do formulário de login o parâmetro de next que nos leva para a próxima página caso a autenticação dê certo:**

<form method**="post"** action**=".?next={{ redirect\_to }}"**> {% csrf\_token %}

<input type**="text"** name**="username"** placeholder**="Username"**/>

<input type**="password"** name**="password"** placeholder**="Password"**/>

<button class**="btn waves-effect waves-light"** type**="submit"** name**="login"**>Logar</button>

</form>

**Para proteger os métodos da *view* e permitir que apenas usuários logados tenham acesso as funcionalidades do sistema, precisamos incluir a anotação @login\_required em cada método do nosso *views.py*:**

**Então com essa anotação caso não haja um usuário logado, é voltado para a tela padrão inicial de login e não abre o template ou executa o método da view.**

**Podemos também deixar para mostrar alguns elementos do HTML da nossa página de acordo com a sessão de login do usuário (se for um usuário root, aparecer tudo do menu, se for um usuário comum, aparecer somente alguns menus):**

1 **{% if request.user.is\_authenticated %}**

2 <li>

<a href=

"{% url 'listar\_usuario' %}">

Listar Usuários</a></li>

3 **{% endif %}**

4  **{%if request.user.is\_staff %}**

5 <li>

<a href=

"{% url 'registrar\_usuario' %}"

>Registrar Usuário</a></li>

6 **{% endif %}**

7 <li>

8 **{% if request.user.is\_authenticated %}**

9 <li>Olá, {{request.user.username}}

</li>

10 <li><a href="{% url 'deslogar' %}"

>Logout</a></li>

11 {% else %}

12 <a href="{% url 'logar' %}"

>Login</a>

13 **{% endif %}**

14 </li>

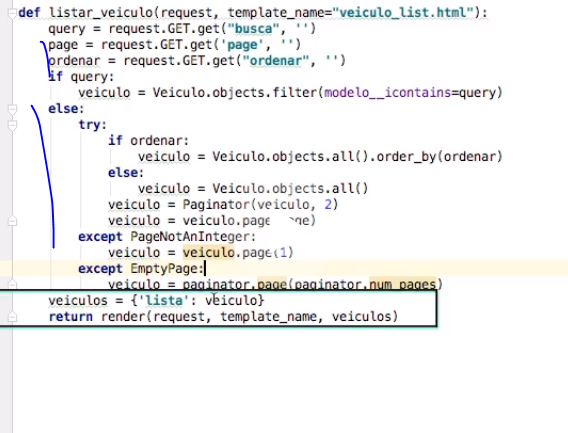
Linhas 1 a 3 - Verificamos se o usuário está autenticado para exibir o menu com a função de listagem de usuários;

Linhas 4 a 6 - O menu para acessar o método de registrar usuário só será exibido caso o usuário logado no sistema seja do tipo *staff*;

Linhas 8 a 13 - Verificamos se o usuário está logado no sistema e, caso positivo, exibimos apenas o menu para a função de *Logout* e uma mensagem de boas vindas. Caso o usuário não esteja logado no sistema, exibimos apenas o menu de *Login*.

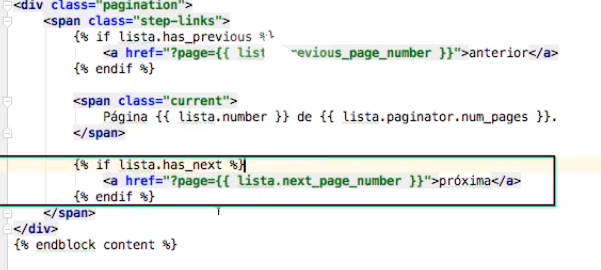
VALE LEMBRAR QUE QUANDO TEMOS UM FORMULARIO E DAMOS UM GET, JOGA O CAMPO PARA A URL COMO UM PARÂMETRO E O SEU VALOR COMO VALOR DO PARÂMETRO NA URL TAMBÉM :D

**Vamos ver como é o recurso de paginação do Django. Pra isso, vamos ver um exemplo que usei para colocar páginação:**



**Para colocar paginação primeiraramente um instancio um um request GET de ‘page’ la em cima, depois nos pegamos os itens de objetos que quero trabalhar, depois aplico a esta lista que peguei um método Paginator onde passo a própria lista e o número de objetos que tem que aparecer por página, depois eu seto com o .page(page) qual é a pagina atual, que é a página própria. Por fim eu coloco dois tratamentos de erros: caso o usuário tentar colocar na URL um parâmetro de página que não é um número, vai para a página 1, e se a página for vazia, listar todas as páginas.**

**Aí no HTML para ter aqueles índices de páginas eu posso colocar o seguinte código:**



**Que basicamente mostra os spans de anterior e next se tiver páginas anteriores ou next, e passo como link desses spans de anteriores e next o método “next\_page\_number” ou o “previous\_page\_number” na URL no parâmetro page=.**

**Para realizar um filtro nos objetos que busquei no banco de dados, podemos usar o comando abaixo:**



**Django Rest Framework**

**Podemos usar o Django Rest Framework para desenvolvimento com Django de APIs Rest.**

**Em uma API Rest temos que criar um Serializer que será responsável para definirmos a forma que os dados será retornados pela API após uma requisição. Com o Django Rest Framework conseguimos definir esse Serializer.**

**Criamos o Model e as configurações de bancos de dados também para trabalhar com nossa API, assim como fazemos somente para o Django normal.**

### Manipulando o arquivo serializers.py

Agora precisamos definir o arquivo responsável por mapear como uma vaga será retornada através das requisições feitas para a API. Sendo assim, no arquivo serializers.py localizado no diretório api do projeto, definimos as seguintes instruções:

**from rest\_framework import serializers**

from .models import \*

**class VagaSerializer(serializers.ModelSerializer):**

**class Meta:**

**model = Vaga**

**fields = ["id", "titulo", "descricao", "salario",**

**"tipo\_contrato", "status"]**

**Linhas 1 e 2:**Importamos o serializer e o model de vaga da aplicação;

**Linhas 4 a 7:**Criamos a classe VagaSerializer responsável por mapear a entidade Vaga do projeto;

**Linha 6:**Definimos que o model a ser utilizado pela classe VagaSerializer é o model de Vaga, definido no arquivo models.py;

**Linhas 7 e 8:**Através do array fields,declaramos todos os atributos da vaga que serão retornados na requisição.

**Para inserirmos dados em um banco de dados via requisição POST, podemos criar dentro de Views a classe seguinte:**

Iniciando o desenvolvimento das funcionalidades da API, o primeiro método a ser construído é o de inserir registros. **Para isso, criaremos uma classe VagaList dentro do arquivo views.py da aplic ção. Essa classe irá comportar todos os métodos que não precisam de um parâmetro para funcionar (GET e POST). Como estamos desenvolvendo o método para inserir registros (POST), vamos defini-lo dentro da classe VagaList como podemos ver abaixo**:

class VagaList(APIView):

def post(self, request):

try:

serializer = VagaSerializer(data=request.data)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data,

status=status.HTTP\_201\_CREATED)

return Response(serializer.errors,

status=status.HTTP\_400\_BAD\_REQUEST)

except Exception:

return JsonResponse({'mensagem': "Ocorreu um erro no servidor"},

status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

**Linha 1:**Criamos a classe que comportará todos os métodos que não precisam de um parâmetro para funcionar;

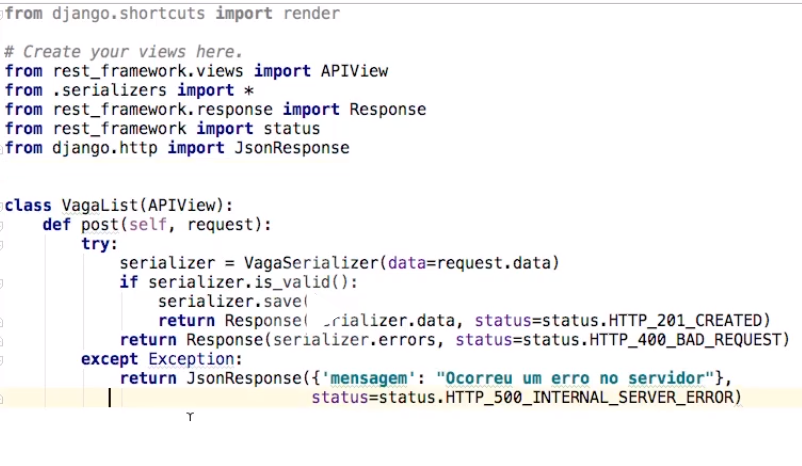
**Linhas 2 a 11:**Definimos o método responsável por inserir registros em nosso banco de dados;

**Linha 4:**Utilizando a classe VagaSerializer, criamos uma nova vaga através dos dados obtidos do corpo da requisição;

**Linhas 5 a 8:**Verificamos se os dados são válidos, ou seja, se estes atendem a todas as regras de validação de uma vaga. Se sim, criamos a vaga no banco de dados e retornamos um código 201, indicando que a vaga foi criada com sucesso;

**Linhas 9 e 10:**Caso os dados da vaga não sejam válidos, retornamos uma lista com todos os erros encontrados e um código 400, indicando que a requisição não foi feita com sucesso;

**Linhas 11 a 13:**Tratamos também o erro interno no servidor adicionando uma exceção ao final do método.



Para que os métodos definidos na classe VagaList sejam chamados, precisamos definir a rota responsável por isso. Sendo assim, veremos nesse vídeo todo este processo.

02:40 min

### Rota da classe VagaList

**Para inserir registros através do método POST, definido no arquivo views.py, precisamos criar a rota responsável por chamar este método.**

**Vimos que a classe VagaList será responsável por armazenar os métodos que não precisam de um parâmetro para funcionar, sendo assim, precisamos criar a rota que aponte para esta classe.**

**Para isso, no arquivo urls.py, localizado no diretório api teremos o seguinte código:**

1. from django.conf.urls import url

2. from .views import \*

3.

4. urlpatterns = [

**5. url(r'^vagas$', VagaList.as\_view()),**

6. ]

**Linhas 1 a 3:**Importamos as configurações de rota e a classe **VagaList** para o arquivo urls.py;

**Linha 5:**Criamos a rota responsável por chamar os métodos definidos na classe VagaList.

**Com isso, definimos a rota que será utilizada em nossas requisições. Vale lembrar que o Django verifica o verbo que estamos utilizando em nossa requisição para determinar qual método será chamado. Sendo assim, ao enviar uma requisição utilizando o verbo POST, o método post da classe VagaList será executado.**

Agora que o método para inserir registros no banco de dados já está funcionando, veremos como desenvolver o método responsável por exibir todas as vagas cadastradas no banco de dados.

### Criando o método get

**Para exibir todas as vagas cadastradas no banco de dados, precisamos criar o método responsável por esta funcionalidade. Para isso, ainda na classe VagaList, definida no arquivo views.py, criaremos o método get como pode ser visto abaixo:**

def get(self, request):

try:

lista\_vagas = Vaga.objects.all()

serializer = VagaSerializer(lista\_vagas, many=True)

return Response(serializer.data)

except Exception:

return JsonResponse({'mensagem': "Ocorreu um erro no servidor"},

status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

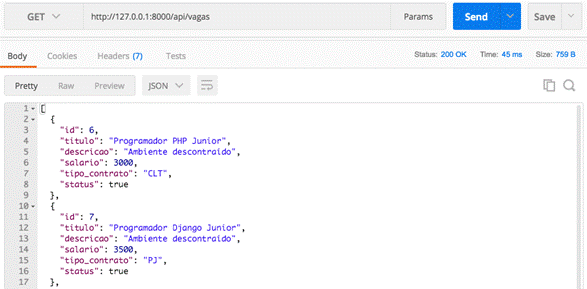
**Linha 3: Obtemos a lista com todas as vagas cadastradas no banco de dados e salvamos na variável lista\_vagas;**

**Linha 4: Utilizando o VagaSerializer, mapeamos os dados das vagas obtidas do banco de dados para serem retornadas através da requisição;**

**Linha 5: Retornamos todas as vagas como resposta da requisição.**

**Linhas 6 a 8: Tratamos também o erro interno no servidor adicionando uma exceção ao final do método.**

**Agora, ao enviar a requisição para a rota /api/vagas utilizando o método GET, serão obtidas todas as vagas disponíveis no banco de dados como podemos ver na Figura 1.**

**Figura 1. Listagem das vagas no Postman**