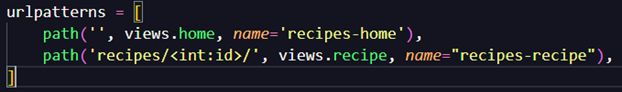
URLS e app\_name para Namespace

1. Modos para chamar URL.

* Modo 1:

No urls.py passamos o name com o que vamos chamar no template



Em seguida podemos chamar no template o name que configuramos:

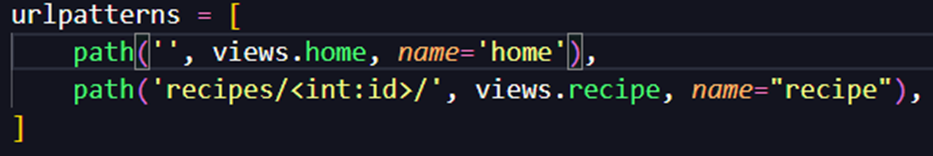


* Modo 2:

No urls.py criamos uma variável como app\_name e um nome correspondente:



Após isso, ainda no urls.py passamos para o name no path o nome correspondente ao que queremos usar no template:



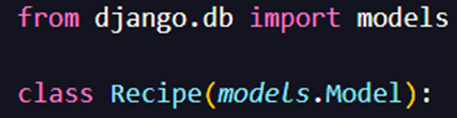
Agora podemos chamar no template de forma mais simples o name que precisamos com <app\_name>:<name> criados:



Models ORM

1. Criando classes (models).

Para criar um model precisamos criar uma classe que herde de “**models.Model**” em **models.py**, onde a classe será considerado uma espécie de tabela da base de dados.



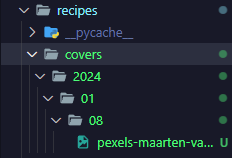
1. Criando campos.

Os campos são criados com a sintaxe de exemplo “**campo = models.CharField(parametros)**”, abaixo estão alguns tipos de campos do django.

1. models.CharField(max\_length=65): Um campo de caractere com um limite máximo de 65 caracteres.
2. models.SlugField(): Um campo usado para armazenar um slug, geralmente uma versão legível da string para ser usada em URLs.
3. models.IntegerField(): Um campo de número inteiro.
4. models.TextField(): Um campo de texto que pode armazenar uma quantidade arbitrária de texto.
5. models.BooleanField(default=False): Um campo booleano que armazena valores verdadeiro/falso. O argumento default=False define o valor padrão como falso.
6. models.DateField(auto\_now\_add=True): Um campo de data que é automaticamente definido para a data atual no momento da criação do objeto.
7. models.DateField(auto\_now=True): Um campo de data que é automaticamente atualizado para a data atual sempre que o objeto é salvo.
8. models.ImageField(upload\_to='recipes/covers/%Y/%m/%d/'): Um campo de imagem usado para armazenar arquivos de imagem. O argumento upload\_to especifica o caminho onde as imagens serão salvas, organizadas por ano, mês e dia.

- Podemos apenas passar o diretório de upload direto na models como demonstrado acima, passando o caminho em upload\_to:  

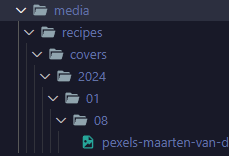

Como ficará a estrutura de arquivos:



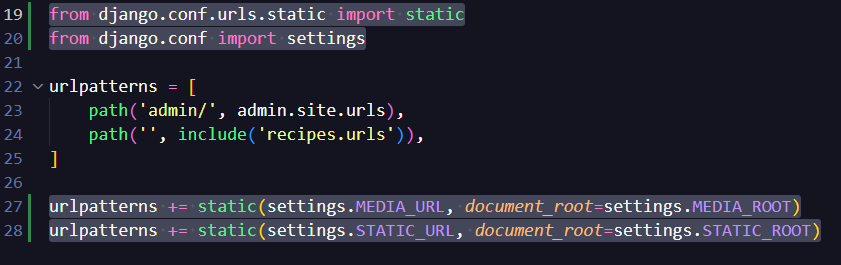
- Mas apesar disso, o ideal mesmo é configurar o caminho no settings.py do Django com os parâmetros MEDIA\_URL e MEDIA\_ROOT, mantendo o caminho definido anterior no upload\_to:



Como ficará a estrutura de arquivos:



- Para acessar ou visualizar a imagem será necessário configurar a urls.py da seguinte forma:



1. models.ForeignKey(Category, on\_delete=models.SET\_NULL, null=True): Um campo de chave estrangeira que estabelece uma relação com outro modelo chamado Category. O argumento on\_delete=models.SET\_NULL especifica que, se a categoria relacionada for excluída, o campo nesta instância será definido como NULL. O argumento null=True permite que o campo aceite valores nulos.
2. Parametros Adicionais

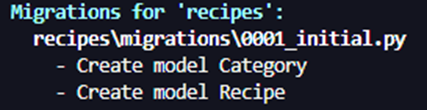
Aqui estão alguns parâmetros adicionais que podem ser utilizados para personalizar os campos mencionados:

1. **models.CharField:**
   * **max\_length**: O comprimento máximo da string (obrigatório).
   * **verbose\_name**: Um nome mais legível para o campo.
   * **help\_text**: Um texto de ajuda exibido ao lado do campo no formulário.
   * **blank**: Se True, o campo pode ser deixado em branco no formulário.
   * **null**: Se True, o campo pode armazenar valores nulos no banco de dados.
2. **models.SlugField:**
   * **max\_length**: O comprimento máximo do slug (obrigatório).
   * **verbose\_name**: Um nome mais legível para o campo.
   * **help\_text**: Um texto de ajuda exibido ao lado do campo no formulário.
   * **blank**: Se True, o campo pode ser deixado em branco no formulário.
   * **null**: Se True, o campo pode armazenar valores nulos no banco de dados.
   * **unique**: Se True, o slug deve ser exclusivo para todos os objetos.
3. **models.IntegerField:**
   * **verbose\_name**: Um nome mais legível para o campo.
   * **help\_text**: Um texto de ajuda exibido ao lado do campo no formulário.
   * **default**: O valor padrão do campo.
4. **models.TextField:**
   * **verbose\_name**: Um nome mais legível para o campo.
   * **help\_text**: Um texto de ajuda exibido ao lado do campo no formulário.
   * **blank**: Se True, o campo pode ser deixado em branco no formulário.
   * **null**: Se True, o campo pode armazenar valores nulos no banco de dados.
5. **models.BooleanField:**
   * **verbose\_name**: Um nome mais legível para o campo.
   * **help\_text**: Um texto de ajuda exibido ao lado do campo no formulário.
   * **default**: O valor padrão do campo.
6. **models.DateField:**
   * **verbose\_name**: Um nome mais legível para o campo.
   * **help\_text**: Um texto de ajuda exibido ao lado do campo no formulário.
   * **auto\_now\_add**: Se True, o campo é automaticamente definido para a data atual no momento da criação.
   * **auto\_now**: Se True, o campo é automaticamente atualizado para a data atual sempre que o objeto é salvo.
7. **models.ImageField:**
   * **upload\_to**: O caminho relativo para o diretório onde as imagens serão salvas.
   * **verbose\_name**: Um nome mais legível para o campo.
   * **help\_text**: Um texto de ajuda exibido ao lado do campo no formulário.
   * **width\_field**: O nome do campo que armazenará a largura da imagem.
   * **height\_field**: O nome do campo que armazenará a altura da imagem.
   * **blank**: Se True, o campo pode ser deixado em branco no formulário.
8. Makemigrations e migrate.

Com as classes e campos criados podemos rodar o comando makemigrations, que será responsável por criar um arquivo para o banco interpretar que geralmente no caminho nome\_do\_app/migrations/numero\_migrations:

>>>python manage.py makemigrations

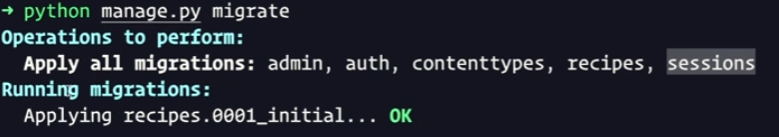
SAÍDA:



Tudo certo e com o arquivo de migrations criado, podemos finalmente rodar o “migrate”, onde será responsável por criar a tabelas e campos na base de dados:

>>>python manage.py migrate

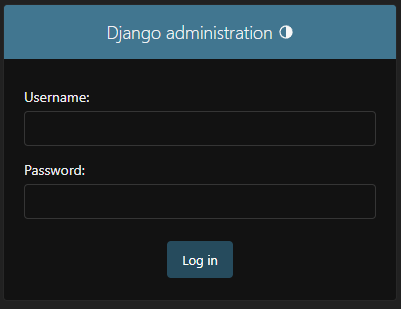
SAÍDA:



Django Admin

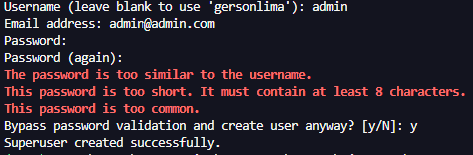
O Django cria um caminho padrão para o sufixo do nosso endereço de servidor sendo “/admin”, ou seja, “127.0.0.1:8000/admin” por exemplo, este local é responsável por uma série de configurações do servidor e possui templates padrões. Muitas empresas usam os próprios templates padrões do Django para gerenciar a organização interna.

Ao acessarmos o Django Administrator, a seguinte tela de login:



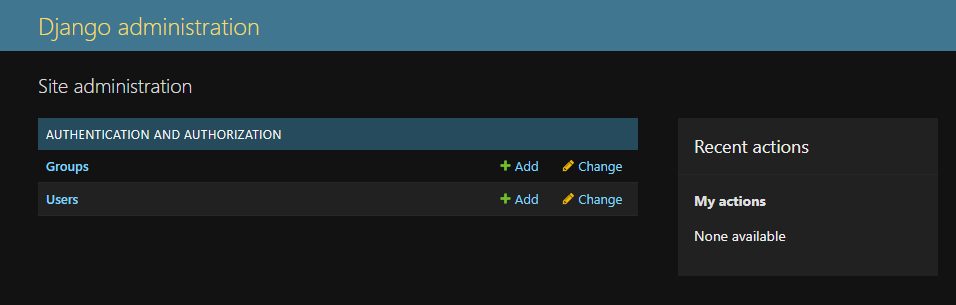
Para que possamos logar, precisamos criar um super user, que por padrão terá direito de acesso as administrações do Django. O comando “**createsuperuser”** é necessário para criar o super user:

>>>python manage.py createsuperuser

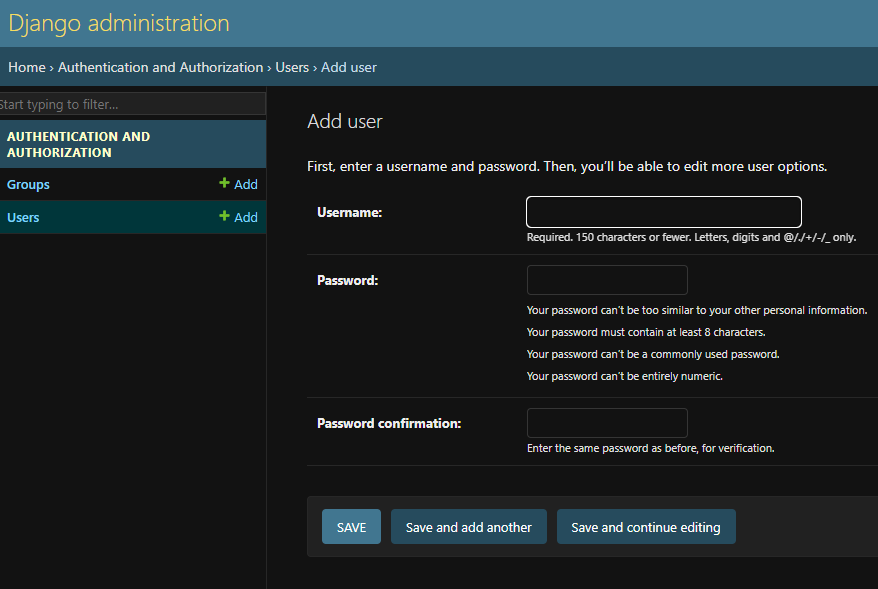
SAÍDA:  


Criei um usuário admin com senha admin apenas para fim didático.

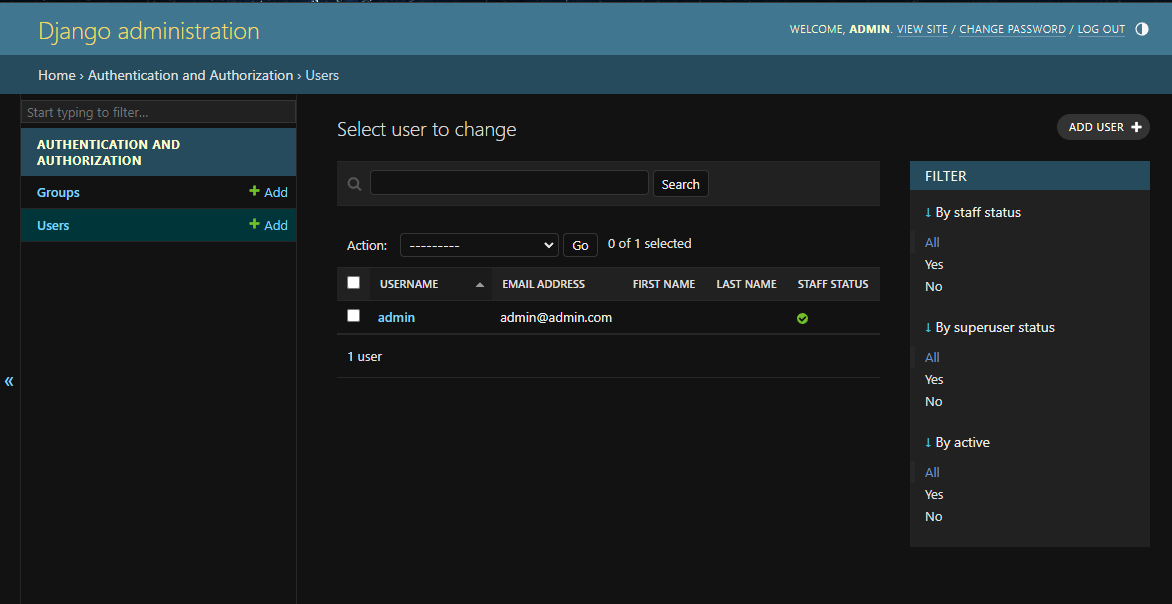
Ao logarmos no Django Administrator teremos acesso as models de Groups e Users, onde Groups é considerado um grupo de permissões de usuário e users são os usuários cadastrados. Para ambas as partes podemos apagar e alterar informações:



+Add:



Change:



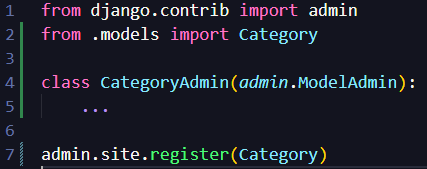
É possível adicionar novas models existentes também na listagem de gerenciamento, dois modos são possíveis no **admin.py** do app você pode usar o decorator como por exemplo: **@admin.register(nome\_import\_model)**,ou o mais simples que é chamando a função **admin.site.register(nome\_import\_model).**

Obs\*: É necessário criar a classe com a model admin.

* **@admin.register(nome\_import\_model)**

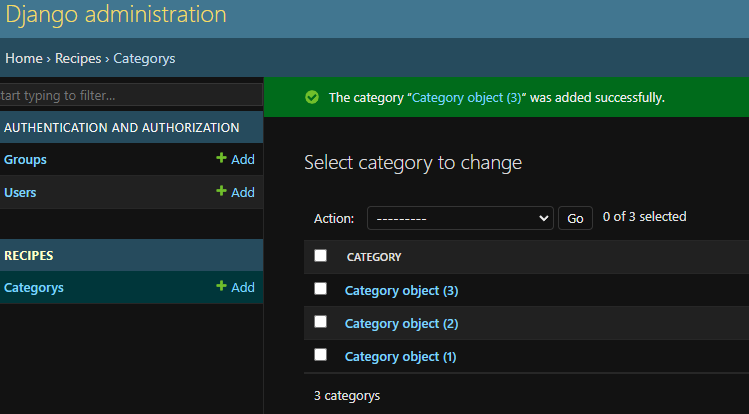


* **admin.site.register(nome\_import\_model)**

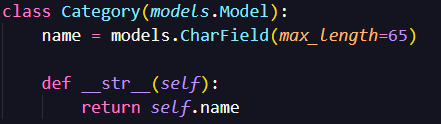


Feito isso, podemos observar que o item na imagem anterior nomeado “Category” passou a aparecer na barra lateral esquerda, onde poderemos fazer cruds para esta model.

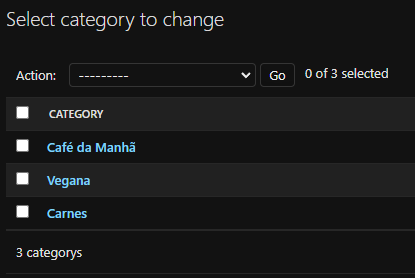
Ao clicar em um item mostrará apenas os campos criados para aquela model.



Na imagem anterior, é perceptível que quando criamos um objeto, ele vem com o nome “Category object(id)” na lista da model. Para solucionar isso, vamos na classe da nossa model e adicionar o método “\_\_str\_\_” a seguir:



Observe que agora podemos ver o nome que foi salvo para a model:



SHELL

O Django Shell é uma interface interativa de linha de comando que permite interagir com o ambiente do Django e realizar operações no banco de dados usando o Django ORM (Object-Relational Mapping). Ele é uma extensão do shell padrão do Python, mas com o contexto do seu projeto Django carregado, permitindo que você execute comandos diretamente relacionados ao seu aplicativo Django.

Para acessar o Django Shell, você pode usar o seguinte comando no terminal, estando no diretório do seu projeto Django:

python manage.py shell

Para listar, criar, atualizar e deletar os objetos da tabela no Django shell, você pode usar os códigos abaixo. Certifique-se de que você já tenha criado a model e tenha executado as migrações necessárias antes de prosseguir.

1. **Listar**
2. Todos os objetos do modelo:

Category.objects.all()

1. Filtrar objetos com base em condições:

Category.objects.filter(name='Nome da Categoria')

1. Excluir objetos com base em condições:

Category.objects.exclude(name='Categoria Excluída')

1. Obter um único objeto com base em condições:

Category.objects.get(id=1)

1. Primeiro objeto do conjunto de resultados:

Category.objects.first()

1. Último objeto do conjunto de resultados:

Category.objects.last()

1. Ordenar resultados com base em uma ou mais colunas:

Category.objects.order\_by('name')

1. Selecionar campos específicos usando values():

Category.objects.values('id', 'name')

1. Remover duplicatas dos resultados:

Category.objects.values('name').distinct()

1. Contar o número de objetos no conjunto de resultados:

Category.objects.count()

1. Criar
2. Inserir um novo objeto:

new\_category = Category(name='Nova Categoria', description='Descrição da Nova Categoria')

new\_category.save()

1. Inserir vários objetos de uma vez:

Category.objects.bulk\_create([

Category(name='Categoria 1', description='Descrição 1'),

Category(name='Categoria 2', description='Descrição 2'),

Category(name='Categoria 3', description='Descrição 3'),

])

1. Utilizando o método create():

Category.objects.create(name='Categoria Criada', description='Descrição da Categoria Criada')

1. Inserir objetos com valores nulos:

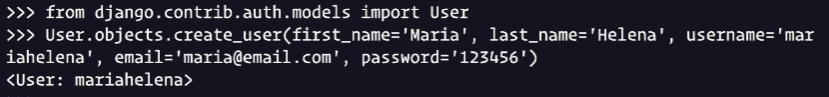
Category.objects.create(name='Categoria Sem Descrição', description=None)

1. Inserir objetos utilizando o método save() após a criação:

new\_category = Category(name='Outra Categoria', description='Outra Descrição')

new\_category.save()

Abaixo um código auxiliar para criar um usuário pelo shell:



1. Atualizar
2. Atualizar um único objeto:
   * Recupere o objeto que deseja atualizar e modifique os atributos:

category = Category.objects.get(id=1)

category.name = 'Novo Nome'

category.save()

1. Atualizar vários objetos de uma vez usando o método **filter()** e **update()**:
   * Atualize todos os objetos que atendem a uma condição específica:

Category.objects.filter(name='Categoria Antiga').update(name='Novo Nome')

1. Atualizar um único objeto usando o método **update()**:
   * Atualize um objeto específico usando condições de filtro:

Category.objects.filter(id=1).update(name='Novo Nome', description='Nova Descrição')

1. Deletar
2. Excluir um único objeto:
   * Recupere o objeto que deseja excluir e use o método delete():

category = Category.objects.get(id=1)

category.delete()

1. Excluir vários objetos de uma vez usando o método filter():
   * Exclua todos os objetos que atendem a uma condição específica:

Category.objects.filter(name='Categoria a Ser Excluída').delete()

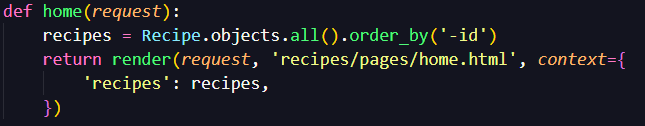
1. Excluir um único objeto usando o método filter() e delete():
   * Exclua um objeto específico usando condições de filtro:

Category.objects.filter(id=1).delete()

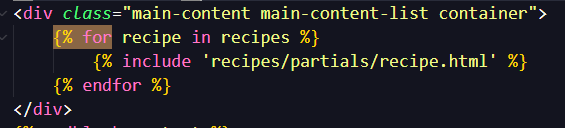
Views

1. Listagem

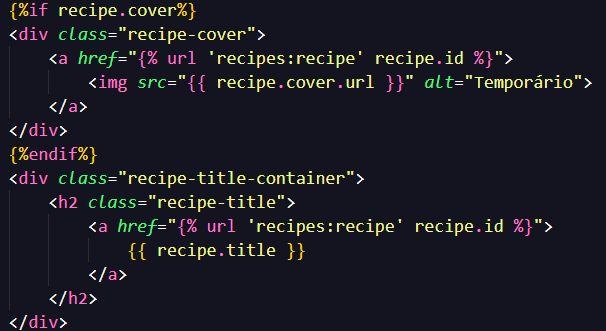
Para listarmos a receita na nossa view precisamos criar a função usando queryset da seguinte forma:



Disponibilizar isso no template será necessário fazer um loop para recipes:



Feito o loop podemos então começar a usar os valores recebidos na nossa página:



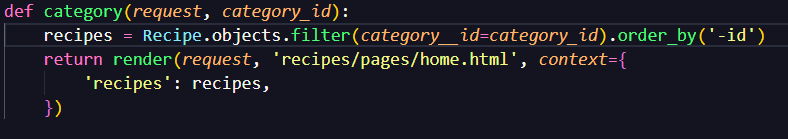
1. Acessar URLS

Podemos acessar algumas coisas de modo simples usando url, por exemplo:

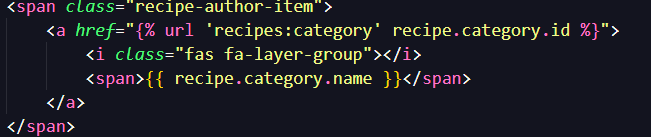
Criamos um **path** em **urls.py** passando o caminho desejado, a função da views e o nome que queremos pra ela:



Feito isso, criamos a função na views.py:



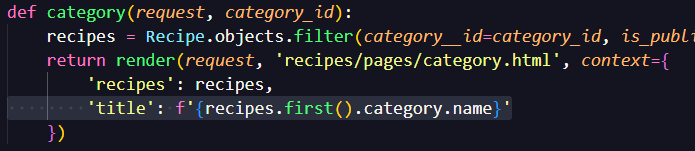
Agora, criada a função passando os dados para o template, podemos adicionar um caminho href usando o parâmetro URL:



1. Título para o head

Podemos também, passar de forma alternada o titulo para o cabeçalho da nossa página, por exemplo, ao usuário acessar uma categoria, será mostrado no cabeçalho o nome da categoria.

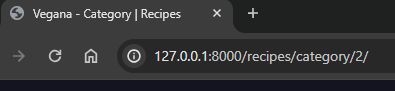
Para isso, precisamos passar os dados de titulo para a nossa função view correspondente a página:



Feito isso, podemos agora ir no nosso template e adicionar o atributo “title” que criamos:



Agora, ao acessar a categoria, temos como titulo no head o nome da categoria:



1. Formatações de Página
2. **linebreaks e linebreaksbr:** Utilizados para converter quebras de linha em tags HTML **<p>** ou **<br>**. **linebreaks** converte quebras de linha em parágrafos (**<p>**), enquanto **linebreaksbr** usa tags **<br>**.

{{ meu\_texto|linebreaks }} {{ meu\_texto|linebreaksbr }}

1. **urlize:** Transforma URLs em links clicáveis.

{{ meu\_texto|urlize }}

1. **truncatewords e truncatechars:** Truncam texto, mantendo um número específico de palavras ou caracteres.

{{ meu\_texto|truncatewords:10 }} {{ meu\_texto|truncatechars:50 }}

1. **date e time:** Formata datas e horas.

{{ minha\_data|date:"F j, Y" }} {{ minha\_hora|time:"g:i a" }}

1. **default:** Define um valor padrão para variáveis nulas.

{{ minha\_variavel|default:"N/A" }}

1. **length:** Retorna o comprimento de uma lista ou string.

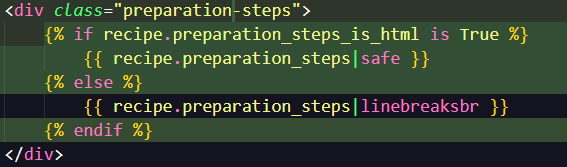
{{ minha\_lista|length }} {{ minha\_string|length }}

1. **floatformat:** Formata um número de ponto flutuante.

{{ meu\_numero|floatformat:2 }}

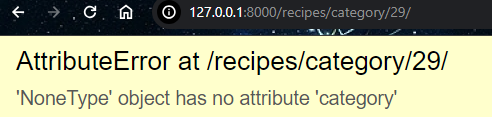
1. **safe:** Marca uma string como segura para ser renderizada como HTML, evitando a escape de caracteres HTML.

EXEMPLO



404 Not Found

Ao tentar acessarmos uma categoria por um ID não existente, recebemos o seguinte erro:

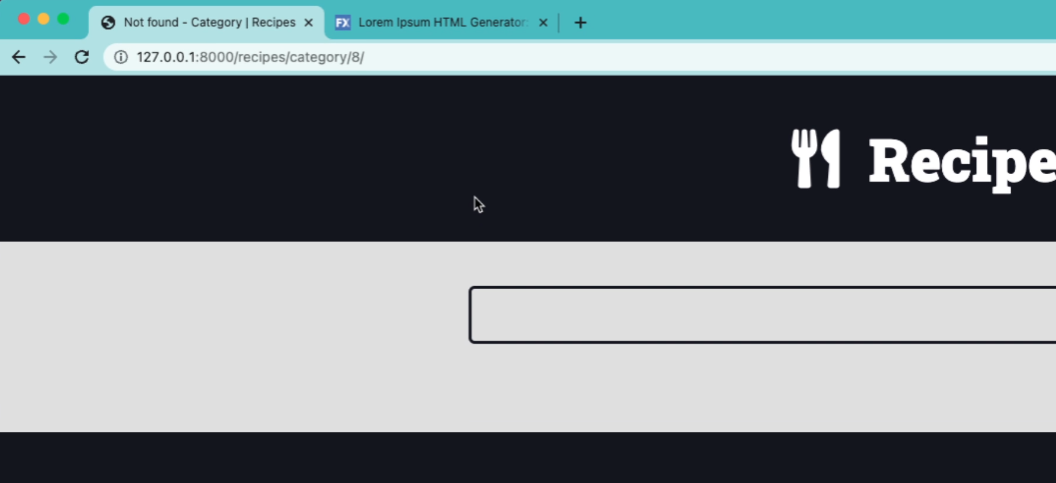


Já que o ID 29 não foi encontrado, podemos contornar o problema com algumas soluções possíveis:

1. getattr()

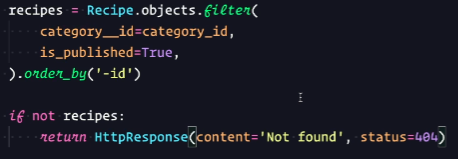
Este modo não é muito recomendado, já que apesar de solucionar a mensagem de erro, ainda recebemos a o request http como 200, onde nesses casos o correto seria receber um 404.



Saída:  


1. HttpResponse()

Este método já é mais recomendado por receber o request 404 do servidor e exige complexidades menores de código no Django. Primeiro temos que importar o **HttpResponse** da seguinte maneira **‘from django.http import HttpResponse’** e no código criamos a seguinte condição na view:



Saída:



1. Http404()

Uma outra maneira no django é utilizar o **Http404**, funciona de forma semelhante ao HttpResponse mas ainda mais simples. Pra isso importamos **Http404** de **‘from django.http import Http404**, agora ao invés de retornarmos um HttpResponse, retornamos um erro Http404:



Saída:



1. get\_list\_or\_404() e get\_object\_or\_404()

O Django também permite estarmos usando os shortcuts(atalhos) **get\_list\_or\_404()** ou **get\_object\_or\_404** importando ‘**from django.shortcuts import render, get\_list\_or\_404, get\_object\_or\_404**’.

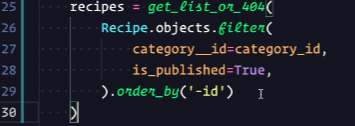
* get\_list\_or\_404():

Neste atalho, ele encontrará uma lista ou retornará uma mensagem de erro, e existem duas maneiras de puxar resultado dele:

* 1. O primeiro modo, é passando a model e os filtros que deseja:



* 1. O Segundo modo, é criar um queryset dentro do atalho:



IMPORTANTE

Ao usar **o get\_list\_or\_404**, precisamos pegar o valor do retorno através do índice se necessário, já que não se trata mais de um queryset e sim de uma lista. Exemplo:

|  |
| --- |
| **ANTES** |
|  |
| **DEPOIS** |
|  |

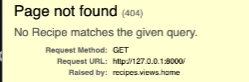
Observe que é necessário substituir o ‘**.first()**’para o índice‘**[0]**’

* get\_objetct\_or\_404():

Neste atalho, ele receberá os dados como um objeto queryset normalmente, porém, automaticamente já nos da a resposta not found:

|  |
| --- |
| **ANTES** |
|  |
| **DEPOIS** |
|  |

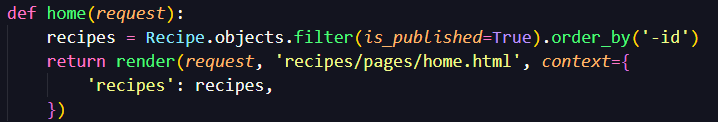
Os dois modos get\_list\_or\_404 e get\_objetct\_or\_404, recebem a saída page not found:



1. {% empty %}

O Django nos oferece um parâmetro chamado **{% empty %}**, onde em casos de não querermos receber o http ‘**Page not found**’ usando os métodos anteriores, podemos carregar a página e mostrar uma mensagem personalizada ao usuário. Exemplo:

View:

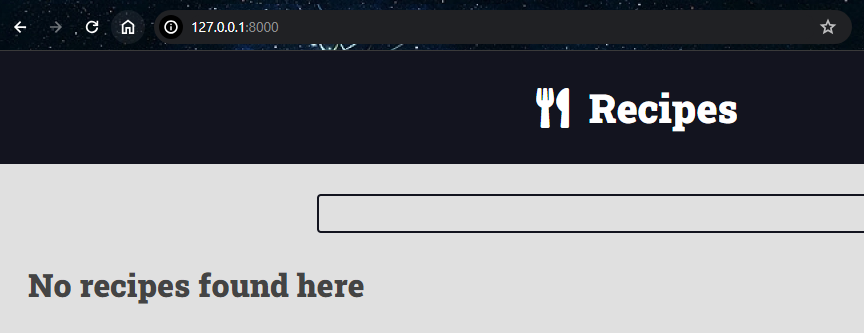


Template:

|  |
| --- |
| **ANTES** |
|  |
| **DEPOIS** |
|  |

Saída:

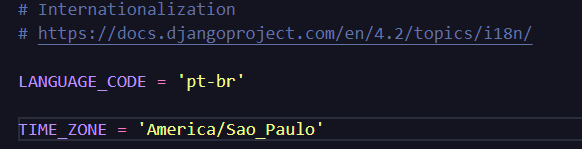
Agora, ao invés de o endereço mostrar uma mensagem de erro ‘**Page not found**’, ele entra na página e nos mostra o que está na condição **{% empty %}**.



Settings

1. Internacionalization

O Django permite que nos settings.py possamos mudar a linguagem e horário para cada país desejado, no nosso caso, vamos mudar o **LANGUAGE\_CODE** para **‘pt-br’** e o **TIME\_ZONE** para ‘**America/Sao\_Paulo**’.



Agora automaticamente o admin fica em português e data e horário é mudado para o horário de São Paulo.

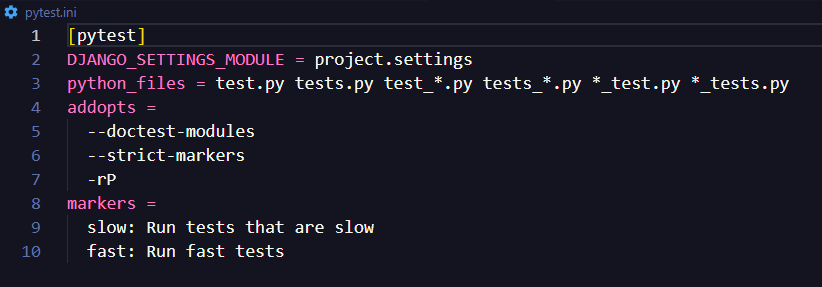
Testes Unitarios

1. Iniciando testes

Com o **ambiente virtual ATIVO** vamos instalar a biblioteca **pytest** necessária para efetuar os testes unitários:

>>> pip install pytest pytest-django

Instalado o pytest devemos criar um arquivo chamado **pytest.ini** no diretório raiz do projeto ou passar os parâmetros do pytest no settings do django. Neste exemplo vamos usar o **pytest.ini**.



DJANGO\_SETTINGS\_MODULE – Aqui passamos o caminho do nosso settings.py do projeto

python\_files – Neste parâmetro passamos os títulos completos ou complementares de arquivos dos quais nosso pytest usará para executar os testes.

addopts – Podemos passar algumas regras para o pytest nesse paramêtros, exemplo:

1. **--doctest-modules:** Com isso, o pytest vai procurar por exemplos de código nos seus arquivos Python e testá-los automaticamente. Por exemplo, se você tem um exemplo na documentação como este:
2. pythonCopy code

>>> 2 + 2 4

O pytest executará esse código e verificará se o resultado é realmente 4.

1. **--strict-markers:** Com essa opção, o pytest fica mais exigente sobre como você usa "marcadores"(markers) nos seus testes. Marcadores são como categorias para os seus testes. Se você tentar usar um marcador que não existe, ou fizer algo errado com marcadores, o pytest vai reclamar.

O fornecido mostra dois marcadores: **slow** e **fast**. Vejamos o que isso significa:

* slow: Run tests that are slow (Executar testes que são lentos):

Este marcador indica que os testes associados a ele são lentos em termos de execução. Quando você executa seus testes, pode escolher executar apenas os testes lentos, se desejar focar em problemas específicos ou otimizações.

* fast: Run fast tests (Executar testes rápidos):

Este marcador indica que os testes associados a ele são rápidos em termos de execução. Se você está iterando rapidamente durante o desenvolvimento e deseja obter feedback rápido, pode optar por executar apenas os testes rápidos.

Ao usar esses marcadores, você pode personalizar a execução dos testes de acordo com diferentes critérios, como desempenho ou tempo de execução para executar apenas os testes lentos:

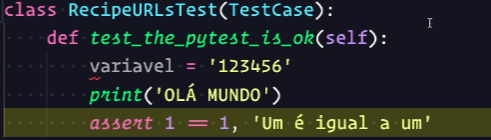
pytest -m slow

Isso executa apenas os testes marcados como slow. Da mesma forma, você pode fazer o oposto para executar apenas os testes rápidos:

pytest -m fast

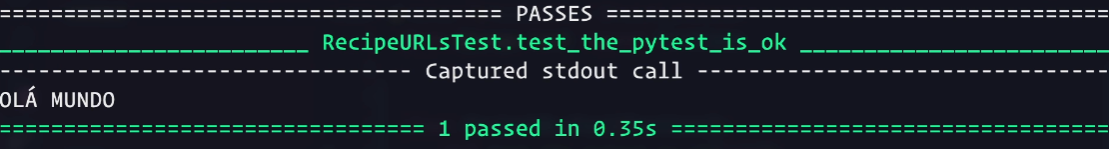
1. **-rP:** Isso significa que o pytest só vai mostrar informações sobre os testes que falharam ou passaram. Se todos os seus testes passaram, você verá uma saída limpa. Se algum teste falhar, você verá uma mensagem indicando qual teste deu errado.

Feito o pytest.ini e passado as métricas em addopts, podemos no diretório de tests.py do nosso projeto Django criar uma classe para testes. O pytest vai automaticamente buscar as funções que iniciam com o que foi configurado no addopts. Segue abaixo um exemplo simples.



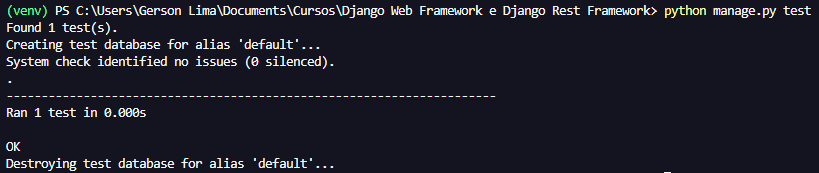
Usando o **runner** do **pytest** temos um retorno mais abrangente e amigável, digitando ‘pytest’ no terminal:

>>> pytest



Mas podemos também utilizar o teste nativo do Django que é o **runner** do **Unittest**:

python manage.py test

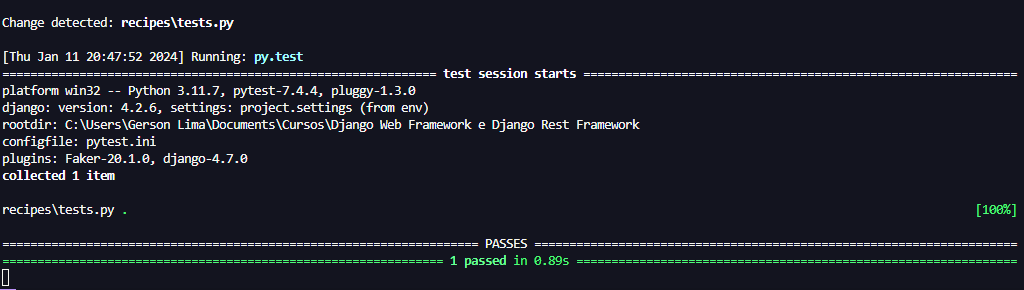


1. **Pytest-watch(ptw)**

Caso queira continuar alterando o projeto e manter o teste automático, basta instalar o ‘**pytest-watch**’ e em seguida digitar ‘**ptw**’, assim não precisará rodar o comando **pytest** a cada alteração.

>>>pip install pytest-watch

>>>ptw



1. Testando URLS do Django.

Agora que pudemos observar como é configurado e garantir o funcionamento do pytest, podemos escrever algumas funções de teste URL.

* **REVERSE()**

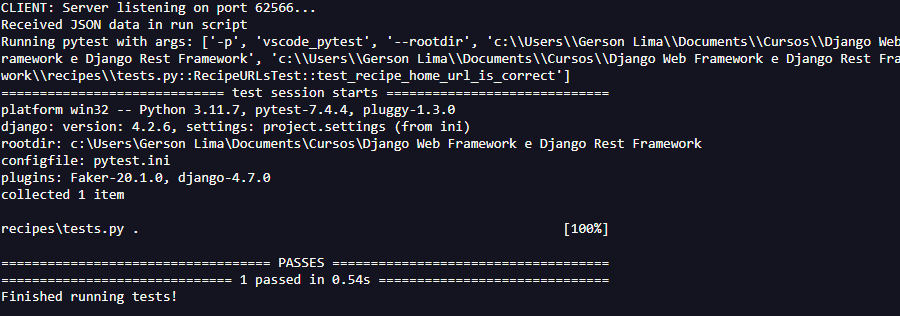
No contexto do Django, a função reverse() é usada para construir URLs reversas a partir de padrões definidos no arquivo urls.py. Ela mapeia uma view para a URL correspondente com base no nome da view ou no identificador do padrão de URL.

* **url sem argumentos:**  
  Vamos seguir criando uma função chamada ‘**test\_recipe\_home\_url\_is\_correct**’, onde irá comparar se o caminho manual da home é o mesmo do path do urls.py:

|  |
| --- |
| **path** |
|  |
| **testes** |
|  |

Observe que para funcionar precisaremos importar ‘**from django.urls import reverse**’ e utilizarmos o método **reverse**, mas uma outra maneira é utilizando o **resolve** que veremos mais a frente.

Ao executar o pytest, se tudo estiver correto teremos a seguinte saída:



* **url com argumentos**

Vamos usar um exemplo onde precisamos listar uma categoria onde neste path o argumento **‘category\_id’** é obrigatório. Seguem as mesmas obrigações do modo sem argumentos, mas com o diferencial de que precisamos passar um dicionário via argumento **kwargs** para o método reverse:

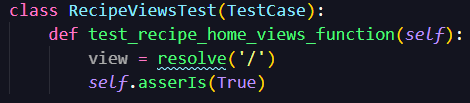
|  |
| --- |
| **path** |
|  |
| **testes** |
|  |
| **saída** |
|  |

1. Testando views

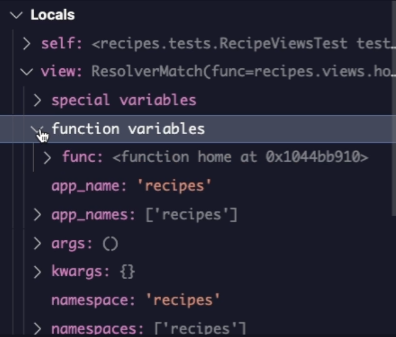
* **RESOLVE()**

A função **resolve()** é usada para resolver uma URL e identificar qual padrão de URL corresponde a uma determinada rota. Ela é comumente usada para obter informações sobre uma rota específica, como a view associada e os argumentos capturados na URL.

Já que o resolve permite que visualizemos informações como a view associada, é exatamente ele que vamos utilizar para criar testes para saber se a view que está sendo chamada é a view correta. Observer o código a seguir:



Criamos uma classe nova para tratar os testes de **views** e em um novo método utilizamos o **resolve** para as informações do que for pego da **view** que é acionada ao acessar a **url** ‘**/**’. Observe na próxima imagem que analisando o que é pego pelo debuger do VS Code podemos ver que nossa variável view recebe uma **function variables** de nome **func**, essa **func** é o retorno em objeto da nossa view, ou seja, é a nossa view **home**.



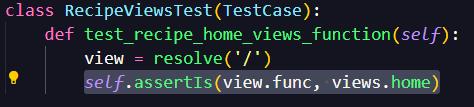
Certo, queremos saber se o caminho ‘/’ de fato está chamando nossa view **home**. Para fazer isso, podemos usar uma asserção **is**, que chega a compatibilidade de dois itens, ou seja, se dois itens são os mesmos.

Primeiros vamos importar as views do nosso app **recipes:**

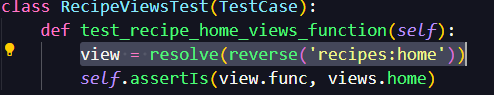
****

Feito isso, vamos usar **assertIs()** para verificar se função que nossa variável **view** criada pegou é a mesma que criamos na **views.py** do nosso app.

* **view sem argumentos**



Caso esteja correto, o teste retornara como um sucesso. Mas o Django nos permite fazer algo ainda melhor, que é invés de pegarmos por ‘/’ que é um caminho de valor Hard coded(estático), podemos pegar o caminho de forma dinâmica.



No nosso **urls.py** definimos o seguinte: app\_name = 'recipes'. Então, podemos colocar o **nome\_do\_app**:**nome\_da\_view** na função **reserve**.

Outra forma estática de pegar é usando get, mas ainda é melhor usar o modo dinâmico. Exemplo estático:



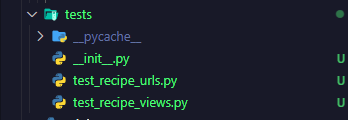
* **view com argumentos**

Para testarmos views com argumentos é bem semelhante ao método que usamos para testar as urls usando resolve.

|  |
| --- |
| **path** |
|  |
| **testes** |
|  |

1. Separando em packages com \_\_init\_\_.py

Apesar do Django permitir que criemos para cada app um tests.py, isso pode ocasionar um script muito grande, o que pode dificultar a manutenção a longo prazo. Sendo assim, podemos criar um package dentro do app para separar os testes, ficando da seguinte forma:



Feito isso, podemos excluir o arquivo tests.py anterior para funcionar, assim, cada classe ficou em um arquivo.py, facilitando manutenções futuras.

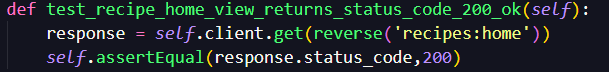
Um comando que podemos digitar para personalizar a saída é o seguinte:

Python manage.py test -v[0|1|2|3]

1. Retorno código http

**- 200**

Para testar se o retorno é 200 usaremos o método **get** com self.**client** na variável **response**, que armazenará os dados de retorno do cliente. Exemplo de código:

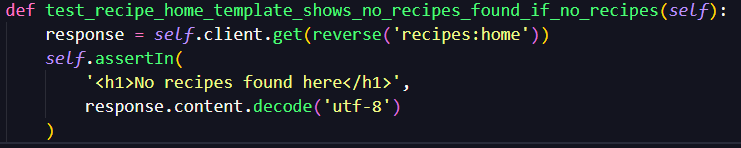


Observe que utilizamos **‘assertEqual’** para comparar o **self.status\_code**, com o valor **200(ok)**. A seguir temos uma lista de response http dentro da variável **response** criada



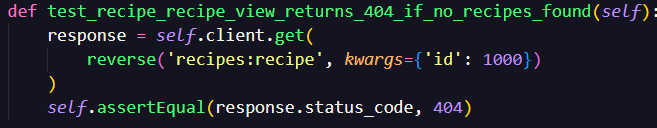
**- NOT FOUND ITEM**

Vimos anteriormente que podemos pegar do **self.client** algumas informações da página, isto inclui as seu código fonte. No projeto simulado, temos que quando nenhum item “**recipe**” é encontrado na **home**, ele retornará um html com o texto ‘**<h1>No recipes found here</h1>**’, sendo assim, usaremos o **assertIn**, para comprar o que há entre os textos do html. Para pode comparar, pegaremos o **response.content** e faremos um **decode**, para formatar o html para **utf-8**. Observe o exemplo:

****

**-404**

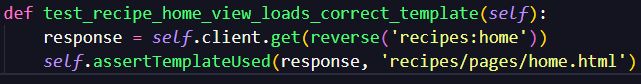
Neste projeto também temos busca por **recipe,** quando o **recipe** não é encontrado ele nos retorna **404**. Vamos testar:

****

Este modo é muito semelhante ao teste de pesquisar a função da view, onde temos que passar a view com seu id de pesquisa, porém. em seguida temos que comprar o **satus\_code**.

1. Nome do template

Para testar o template usamos o método **self.assertTemplateUsed,** exemplo:



1. Carregamento Items no Template

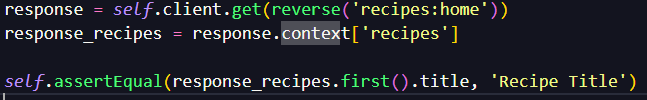
Para verificarmos se está sendo carregado os itens do template, vamos precisar simular a criação de um item (no caso uma recipe) para que seja carregado no teste.  
Primeiro temos que importar as model desejadas para criar o item.



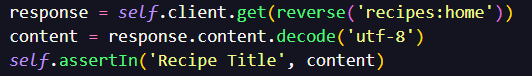
Em seguida, podemos criar os itens necessário para dar continuidade na validação de carregamento de itens:



Logo após criado o recipe, é possível testar o item criado e se a view consegue pegar os dados usando **context**.



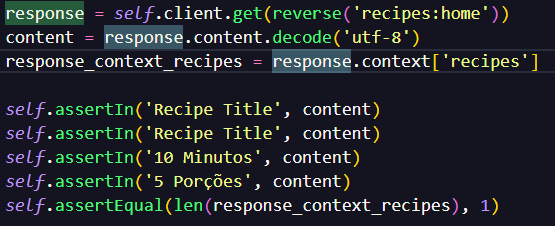
Mas além disso, podemos também colocar como teste se o item criado foi recebido e carregado pelo template da seguinte forma usando **content**:



Basicamente, o **content** retorna uma string da pagina html, como se fosse um inspecionar elemento:



Para garantir a saída do template, podemos usar o **context** visto anteriormente:

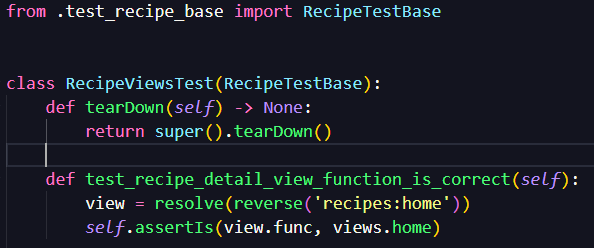


1. setUp() e tearDown()

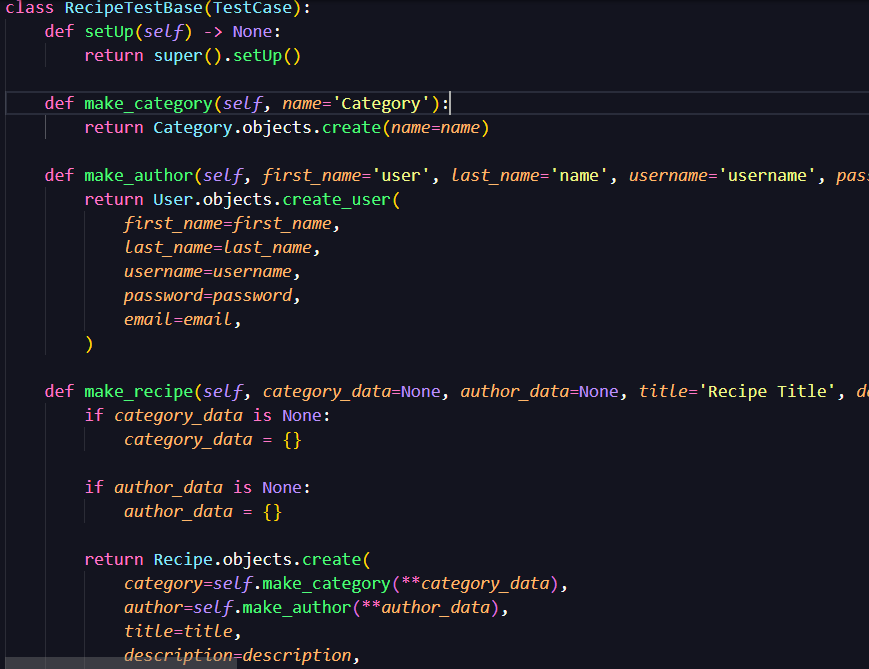
A solução vista no tópico anterior[8] é pouco recomendada, pois torna o código pouco pratico para reutilização da criação dos objetos, sendo necessário reescrever os objetos para cada método.  
Tendo em vista esse caso, um grande aliado na solução é o método **setUp()** e **tearDown().** O **setUp** sempre será executado em primeiro lugar ao chamar a classe. Sendo assim, primeiro vamos colocar nosso queryset em uma classe separada:



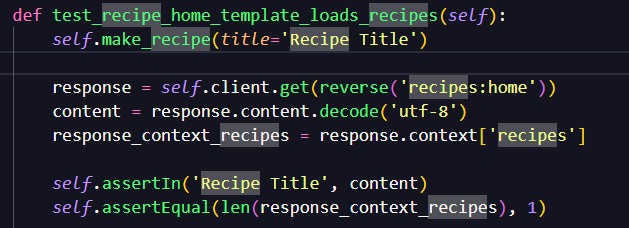
Agora, vamos fazer com que nossa classe de tests de views e template herde da classe RecipeTestBase criada na imagem anterior.



Infelizmente para outros testes que de Not Found por essa solução apresentara problemas, pois sempre criará um item para ser lido. Sendo assim vamos abordar uma forma mais inteligente e dinâmica.  
Primeiro, vamos separar cada criação de objeto por método dentro da classe da seguinte maneira:



Agora, podemos de uma forma muito mais limpa criar o recipe e testar se está sendo carregado no template usando a função **make\_recipe** criada na última imagem:



Observe que passamos o titulo dentro da função, mas outros parâmetros podem ser passados para criar o item de acordo com a necessidade.

Confira as alterações em: <https://github.com/Gersonline/curso-django-projeto1/compare/7ed63520effacc0f9701d50377aa6b0e96cac0a9...ed6899160951ad239720d827fc092f3d599495b3>

1. @skip e self.fail()

Caso necessário, é possível forçar um erro no test ou pular o test.   
  
**@skip()** – Serve para pular o test, e pra ele precisamos importar **‘from unittest import skip’** e antes do método ou classe digitar **@skip(‘Mensagem do porque pular’):**



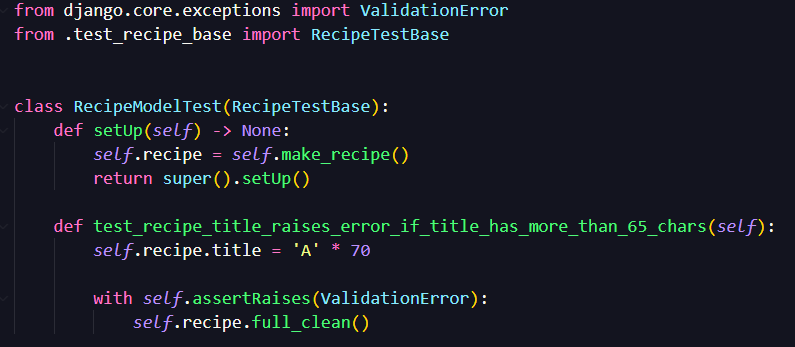
**self.fail()** – Nele forçamos um erro e imprimimos uma mensagem:



1. Testando Models

**- Solução 1**

Apesar do Django já possuir testes internos, existem exceções para serem validadas. Por exemplo, vamos colocar um test que limita nossa model a 65 caracteres.  
Para isso, importaremos ‘**from django.core.exceptions import ValidationError**’, utilizaremos ‘**assertRaises()**’ e o método **full\_clean()**, responsável por validar os objetos.

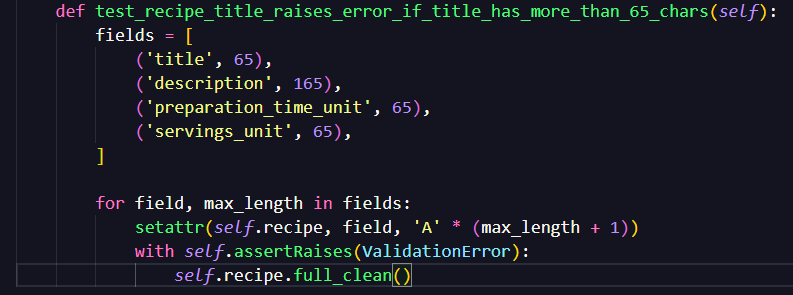


**Entendendo:**

Se fizermos **self.recipe.save()** infelizmente não funcionará, pois o Django não nos retorna nos testes a mensagem de erro do save, dando a impressão de que o recipe foi salvo.  
Usar o **assertRaises** confirma se o **ValidationError** retornado pelo **full\_clean()**  é de fato um erro.  
Então se por acaso mudarem no model o valor para ‘max\_length=650’, ele nos retornará um erro, pois a função verifica se o valor inserido de fato passou do limite.

**- Solução 2**

É possível também testar de forma mais dinâmica campo por campo.



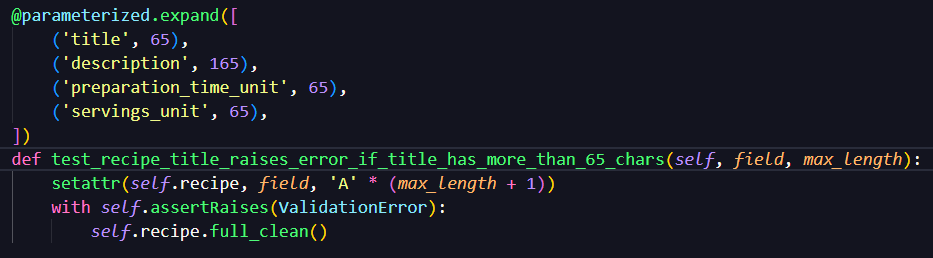
**Entendendo:**

Dessa vez criamos uma lista como nossos campos e seus tamanhos máximos, e percorremos um por um, atribuindo valor no campo através do **setattr()** e seguindo o fluxo padrão do **assertRaises** da solução 1.

**- Solução 3**

Esta forma é a mais recomendada quando não utilizar **pyTest**, pois permite diferenciar cada campo que ocasionou o erro, e não apenas 1 erro do método da classe. No Django padrão utilizamos **unittest**, o que nos retorna saídas diferentes.  
Então vamos lá, o que precisamos é utilizar a biblioteca parameterized, que nos permite separar cada item da tupla como um teste e importa-la dentro do arquivo ‘**from parameterized import parameterized’**.

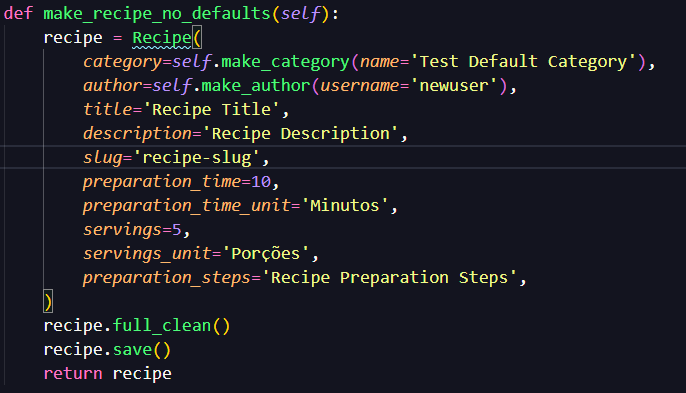
>>>pip install parameterized



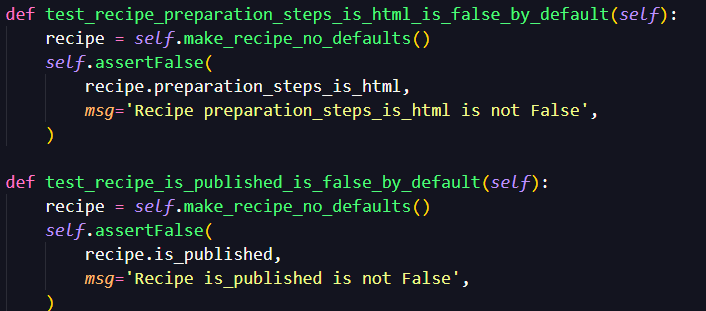
1. Testando Default do Model

É possível que queira também treinar o default, para isso vamos utilizar **assertFalse()** na criação de um recipe para saber se de fato o padrão foi salvo como false.

Primeiro vamos criar uma função que cria um recipe onde o campo **is\_published e preparation\_steps\_is\_html** são **false** por padrão.



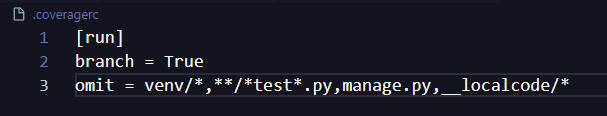
Agora podemos criar as duas funções de teste que chamam a função que cria recipe e testar o objeto.



1. Biblioteca coverage para testes

Basicamente, o **coverage** é uma biblioteca para cobertura de testes, onde nos mostra o que foi testado e o que falta ser testado na nossa aplicação.

O primeiro passo, assim como no pytest é criar um arquivo de configuração para o **coverage**. Na raiz do projeto vamos criar um arquivo com o nome **.coveragerc** e incluir os seguintes parâmetros:



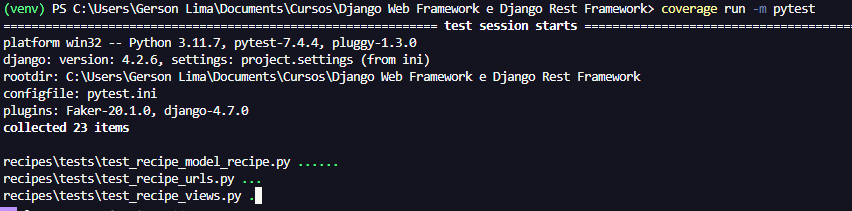
Agora o vamos instalar a biblioteca

>>>pip install coverage

Agora para iniciar o coverage, podemos escolher entre o **pytest** ou o **unittest** do Django.

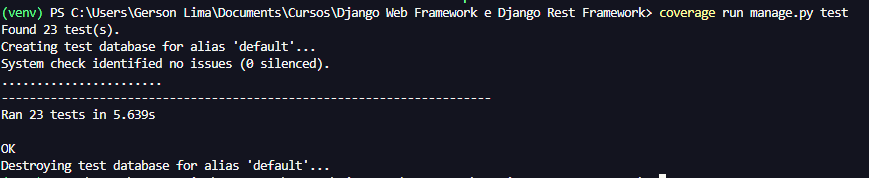
**Pytest:**

>>>coverage run -m pytest



**unittest:**

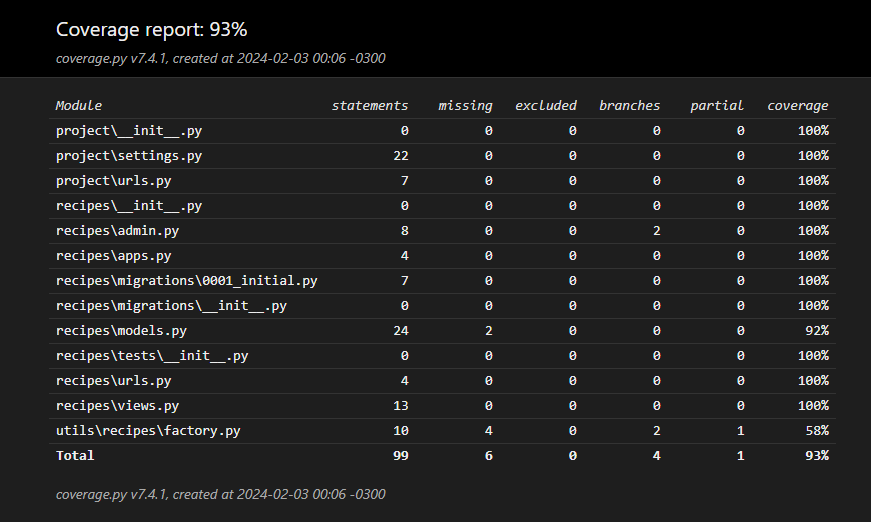
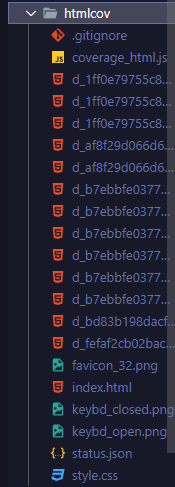
>>>coverage run manage.py test



Legal, agora para visualizarmos os status da nossa cobertura de testes, vamos usar o comando:

>>>coverage html

Em seguida ele criará uma pasta com o nome **htmlcov onde poderá ser visualizado em um html.**



Request.GET e TDD

Antes de iniciarmos, primeiro vamos fazer algumas preparações e correções.

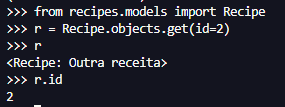
1. Duplicar itens usando shell

É simples e importante que para facilitar o uso do Django, saber manusear o shell do python.  
No caso em questão, usaremos ele para duplicar as receitas do no nosso projeto para fazermos alguns testes e não termos que cadastrar uma a uma.   
Sendo assim, prosseguimos rodando o comando no teminal:

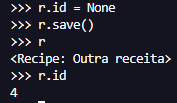
Python manage.py shell

Agora, com acesso ao terminal do Django, podemos trabalhar.

**1º -** Vamos importar uma receita.



**2º -** Agora que temos um objeto receita e um ID, se dermos um **r.save()** vamos alterar a receita. Maaas... se colocar que r.id(ou r.pk) é igual a None e dermos um **r.save()**, o Django pensará que é um novo objeto por conta do auto incremento, já que não encontrou ID.



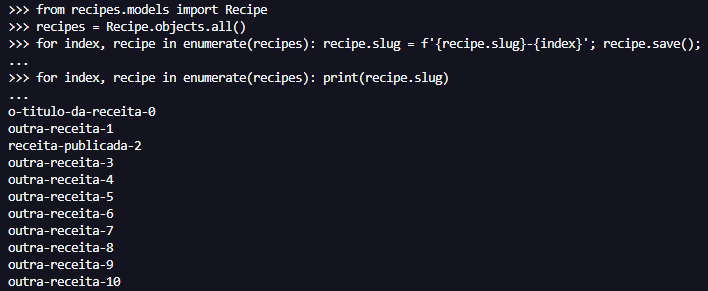
Observe que agora ele trouxe um novo id, pois é um novo objeto.

**3º -** Vamos fazer um loop para não precisar repetir esse processo manualmente.



1. Unique SlugField

O nosso campo slug precisa ser unique, o problema é que se mudarmos agora na models para unique nos retornará um erro, pois o ultimo processo repetiu também os campos de slug. O que temos que fazer primeiro então, é no shell alterar os valor desses campos da seguinte forma:



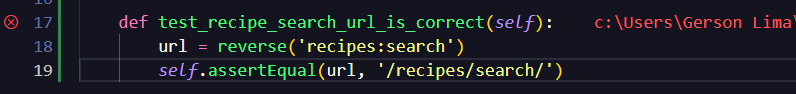
Feito isso, agora podemos setar o campo slug como **unique=True** e criar e rodar a migration:



Criando uma nova URL para Search com TDD(Test Driven Development)

Em TDD temos como metodologia desenvolver com base nos testes, ou seja, primeiro criamos o teste e só então damos inicio ao desenvolvimento daquela função.

Neste exemplo, vamos criar um test para a busca de receitas em um site. E como dito, vamos primeiro criar nossa função de teste.



Ao tentar executar a função dará erro, logicamente, pois ainda não foi criada a URL para Search, mas essa é intenção, desenvolver em cima do teste.

**Antes de prosseguirmos**, é importante entendermos um **Ciclo do TDD**, a ordem é a seguinte **RED > GREEN > REFACTOR.**

**- RED**: Primeiro desenvolvemos o teste onde nos trará um erro pois a solução não foi criada.

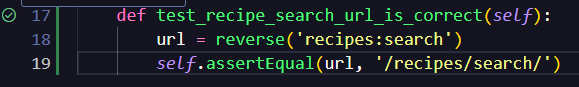
**- GREEN**: Em cima do teste desenvolvemos uma solução mais rápida possível para passar o teste.

**- REFACTOR**: Como no GREEN pode ser usado algo não usual, mas sim apenas funcional para passar no teste, após passada a etapa Green, refatoramos o código para ser usual e não apenas funcional para o teste.

Entendido isso, podemos dar continuidade.

Vamos criar a nossa URL, mas, criaremos algo funcional apenas para passar no teste e mais para frente refatoramos.



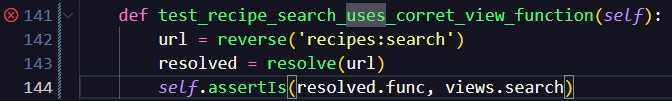


Criando a Search view function com TDD

No tópico anterior, criamos um teste e uma url, mas para passar no teste criamos uma função lambda no path.

Agora, vamos de fato criar a nossa view function, usando os princípios de TDD.

**- RED**: Criamos o teste.

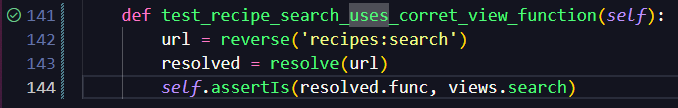


- **GREEN:** Criamos a função para passar no teste.



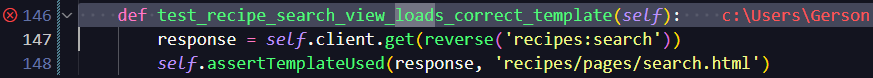
- **REFACTOR:** Aqui precisamos fazer o refactor da url anterior para chamar a função Search.





Indicando para a view retornar o template correto com TDD

**-** **RED**: Vamos criar o teste para ler o template



- **GREEN**: Vamos passar um return http para a view Search e criar um arquivo search.html.



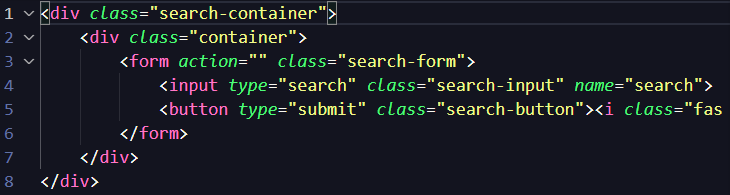


Redirecionamento Erro 404

Aqui faremos um redirecionamento para um erro 404 quando a busca não for encontrada.

Importante termos como base o funcionamento de que as pesquisas na web funcionam com querystring, por exemplo... Ao consultarmos ‘Bolo’ teremos uma URL semelhante a isto <www.meu-site.com/?search=Bolo>

Certo, dito isso, vamos ao nosso código da barra de busca... temos o seguinte formulário para barra de busca:





Observe que no nosso <form> não temos um method, que pode ser ‘GET’ ou ‘POST’. Por padrão se não instanciarmos o method a pagina entende que ele é um ‘GET’.



Atente-se também que o que está como ‘name’ no nosso input é o que aparecerá na URL, por exemplo:

* Alterando name para ‘q’:



* Como fica a URL:



Legal entendemos oque faz o method do form... mas e o action Simples, o action é responsável por passar literalmente as ações que será feita após a requisição, seja ela ‘GET’ ou ‘POST’.

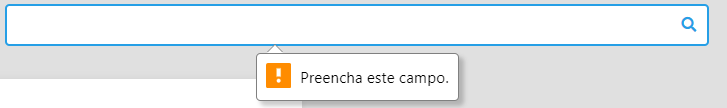
Então vamos dizer para nosso action que o mesmo nos redirecionará para nossa URL ‘search’ .



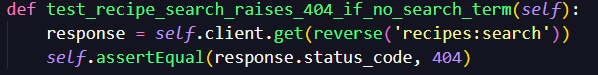
E ao fazer a pesquisa iremos para a seguinte URL:  


Agora nós fomos redirecionados para ‘recipes/search’ recebendo a querystring ‘Bolo’. Caso não passamos um action a query string seria enviada diretamente para nossa página atual.

\*IMPORTANTE: Dessa forma, mesmo que o usuário não digite nada no nosso **form** ele ainda será redirecionado para a página de busca. Para que isso não aconteça podemos passa para o input o parâmetro ‘required’:  
  

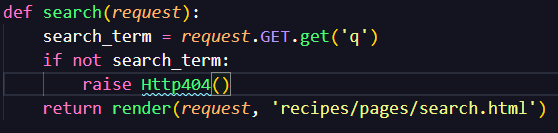

Agora, ao tentar fazer uma pesquisa sem digitar nada receberemos o próprio HTML nos mandara um mensagem e não acionara o **action**:  


Finalmente vamos então para tratativa de ERRO 404, mas antes vamos fazer um test abordando a metodologia TDD.



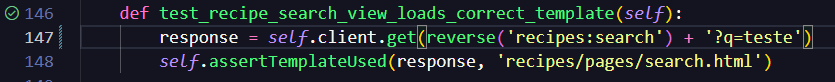
Essa função de TDD irá verificar se o retorno da página recebeu de fato um **status code** 404. Porém, mesmo que não haja nada ele receberá um 200, pois a pesquisa mesmo que vazia, ainda é efetuada quando digitada direto na URL.   
Então faremos o seguinte, usaremos o **request.GET** na view **search** para verificar se existe algum valor.





Note que usamos **request.GET.get(‘q’)** e não apenas **request.GET()** pois caso não fosse passado o parâmetro ‘q’ receberíamos um erro 500(Falha no servidor). E após feito isso, colocamos a regra de que se não for passado um termo de busca receberemos da nossa pagina o erro http 404.

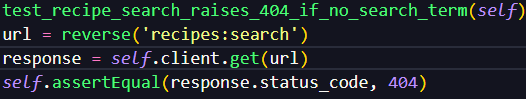
Atenção: Outros testes falharão agora que tratamos o retorno de HTTP404, agora para a busca ser valida ela precisará conter o parâmetro ‘q’ com algum valor. Logo fica assim o nosso teste de pesquisa colocando ‘?q=teste’:



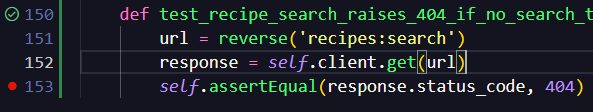
\*EXTRA:

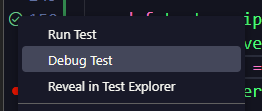
Podemos fazer o seguinte para testar se a busca trará o resultado correto.

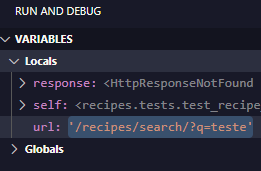
1- Criamos uma url



2 - Colocar o brakepoint na função de teste:

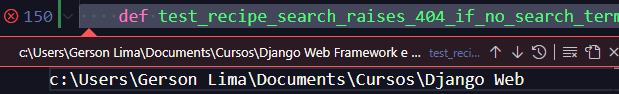


3 – Com o botão direito no play do test selecionar ‘Debug Test’:  


4 – No parâmetro ‘url’ que aparecera na lista de debug colocar o caminho com a busca desejada:  


5- Agora damos um continue:





E a saída retorna um erro, o que no caso desta função é o esperado, já que ele retorna ‘200’ e não ‘400’ como a função de teste espera. Testamos o teste.

Filtrando value de pequisa

Temos alguns pontos a serem considerados no momento de pesquisa, podemos alterar o title da pagina para mostrar a busca no cabeçalho do navegador e também impossibilitar que o usuário envie apenas um espaço para a pesquisa.

* Title

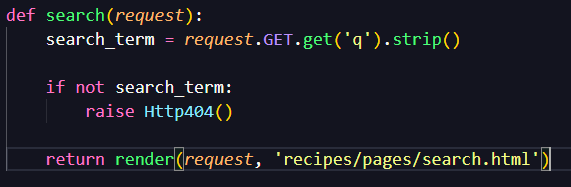
Em um arquivo novo (aqui nomeado Search.html na pasta pages) vamos colar uma copia da nossa home. E em seguida passamos o valor de uma variável que será enviada pelo contexto do Django. No caso foi adicionado no title ‘{{ page\_title }}’.

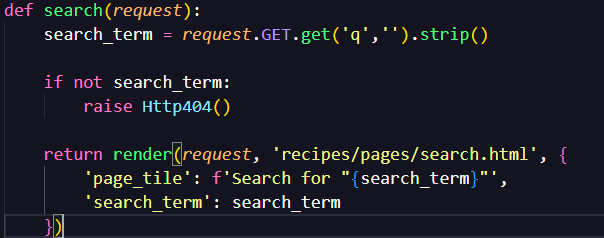


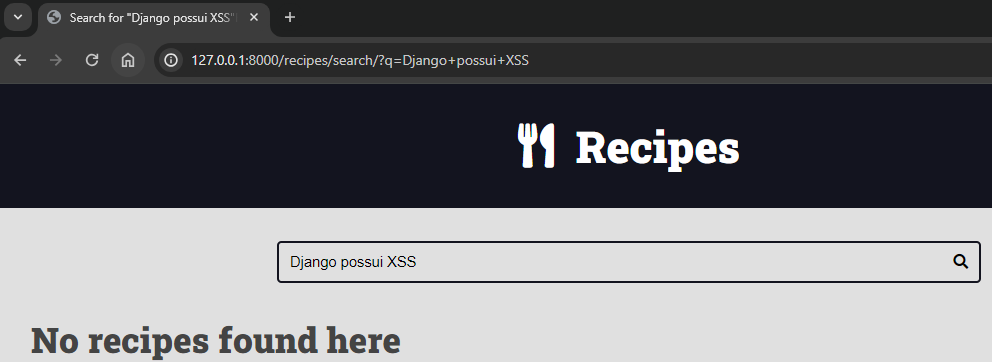
Feito isso vamos também passar um valor para nosso input de pesquisa, para que assim que for efetuada a pesquisa não seja perdido o valor do input. No caso chamaremos a variável de ‘search\_term’.



Agora para que as página de Search receba os valores, fazermos a seguinte alteração no nosso código da view:

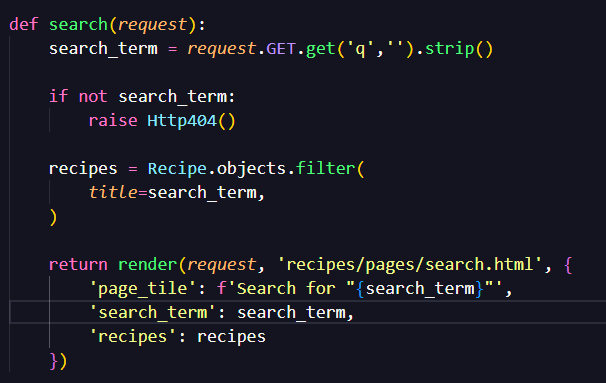
- Antes:  


- Depois:  




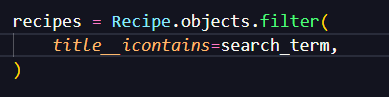
Observe que enviamos ao Django dois valores dentro do ‘render’, o terceiro vetor separado por virgula é destinado aos valores ‘context’, ou seja, são aquelas variáveis que serão renderizadas no Template. Outro ponto importa é que agora a variável ‘search\_term’ não receberá valor apenas com um espaço com o ‘.get('q','').strip()’.

Agora, podemos pesquisar os itens de acordo com a busca, para isso vamos na nossa função de busca “Search”



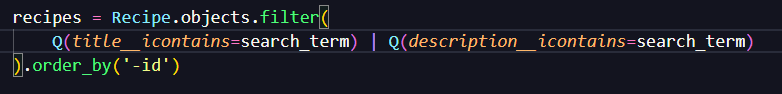
Certo, o problema deste modo é que o “title=” mostrará apenas o que for exatamente igual o que for digitado. Para isso temos o \_\_contains e o \_\_icontains:

* \_\_Contains pode dar inconsistências com letras maiúsculas e minúsculas
* \_\_Icontains traz independente do maiscula ou minuscula

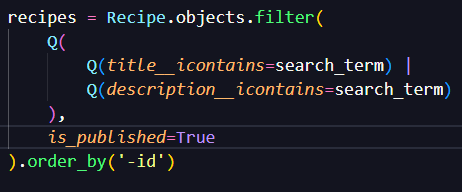


Podemos também ordenar nossa listagem com order\_by(‘-id’) e adicionar um filtro a mais, mas para adicionar um filtro a mais precisaremos importa Q e usar |. Observe:





Legal, mas agora temos um erro de pesquisa perigoso, devemos trazer apenas as receitas publicadas e dessa forma ele ignora e traz as receitas não publicadas. Para resolver isso fazermos o seguinte:



Paginação

O Django possibilita uma paginação automática para o retorno de informações no nosso Template. Para fazer isso seguiremos os seguintes passos:

1. Views

1. Importamos o Paginator no nosso views.py:



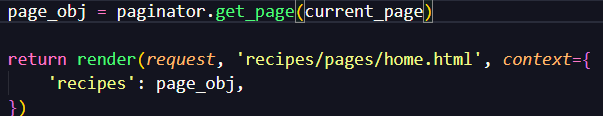
2. Depois, pegaremos o número da página passada na URL, onde ‘page’ será o parâmetro que contém o número da página, e se não houve nada ele será 1:



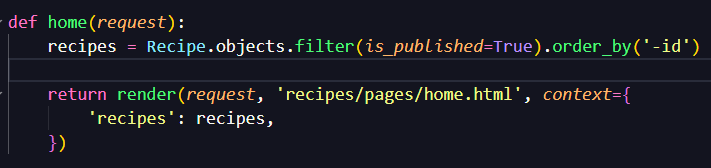
3. Agora vamos chamar o Paginator, e passar o primeiro parâmetro o resultado da nossa queryset “recipes” e o segundo parâmetro será a quantidade de itens por página:



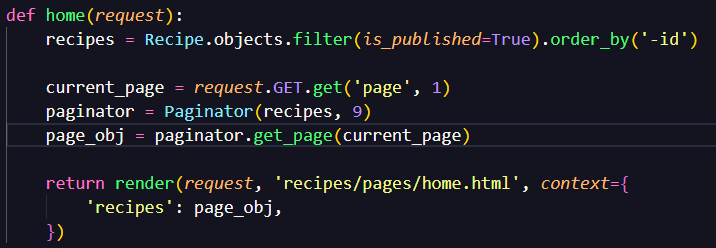
4. Tudo isso se tornará um obj a ser renderizado pela pagina, então ao invés de mandarmos o recipes para o render, vamos fazer o seguinte:



**Antes:**



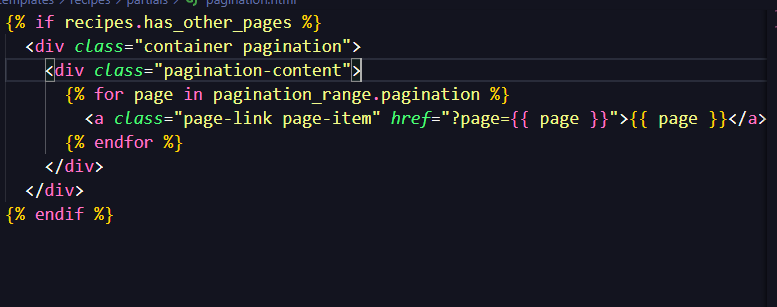
**Depois:**



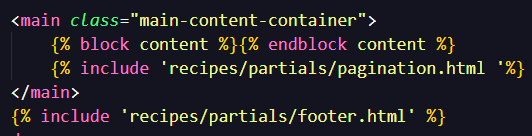
1. Templates

Feito a parte do views, vamos fazer só mais uma alteração nela e depois vamos então implementar a visualização para navegação no tempate. Para isso, vamos criar um arquivo .html que chamaremos de pagination.html e dar o seguinte código:





E incluímos no home (onde queremos mostrar):



Observe, a função makepagination range é encontrada no seguinte commit: <https://github.com/Gersonline/curso-django-projeto1/commit/9a7005fcf6ffe30a1ba1760b3b2d78111c12e221>

Estilização do pagination: <https://github.com/Gersonline/curso-django-projeto1/commit/df767e948403bc4b50f1b646db83ad010c5cceec>

Referencias

<https://www.lipsum.com/feed/html>

<https://www.pexels.com/pt-br/procurar/cooking/>

<https://www.webfx.com/web-design/html-ipsum/>