Tecnologia em Análise e Desenvolvimento de Sistemas
Tópicos Especiais em Tecnologia da Informação
Prof. Rafael Henrique Dalegrave Zottesso

Programação para Web com Python e Django

Documentação dos Sistemas de Autenticação do Django
https://docs.djangoproject.com/en/2.2/topics/auth/default/

Nesta etapa, vamos relacionar nossos modelos com a classe "User" do Django para podermos separar os objetos por usuário, de modo que um usuário tenha seus próprios registros no nosso sistema e que um usuário não acesse os registros de outros usuários.

Importe o User no models.py para poder criar chave estrangeira com ele
from django.contrib.auth.models import User

Crie um atributo na sua classe, neste caso criei no Animal

```
class Animal(models.Model):
    tipo = models.ForeignKey(Tipo, on_delete=models.PROTECT)
    ...
    telefone = models.CharField(max_length=17)

    usuario = models.ForeignKey(User, on_delete=models.PROTECT)

    def __str__(self):
        return "{} - {}/{}".format(self.nome, self.raca, self.tipo)
```

Execute o makemigrations para preparar seu banco de dados com as modificações

```
python manage.py makemigrations adocao
```

```
You are trying to add a non-nullable field 'usuario' to animal without a default; we can't do that (the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
 2) Quit, and let me add a default in models.py
Select an option:
```

Esta mensagem indica que você está alterando uma tabela no banco que já tem registros. Como não falamos que ele pode ser "blank" e "null" você precisa fornecer um valor padrão. Por isso, vamos na opção "1" e depois digitar um valor qualquer para ser preenchido na nova coluna "usuario", por exemplo "0".

Execute o migrate para atualizar o banco

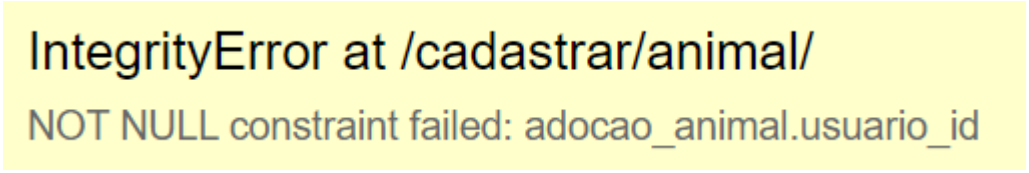
```
python manage.py migrate
```

```
 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do e.g. timezone.now
Type 'exit' to exit this prompt
>>> 0
Migrations for 'adocao':
  adocao\migrations\0005_animal_usuario.py
    - Add field usuario to animal
PS C:\Users\zotte\Área de Trabalho\2019-TADS-Topicos-Especiais\Django> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, adocao, auth, contenttypes, sessions
Running migrations:
  Applying adocao.0005_animal_usuario... OK
PS C:\Users\zotte\Área de Trabalho\2019-TADS-Topicos-Especiais\Django>
```

Inicie o servidor novamente e acesse a página para cadastrar um animal

```
python manage.py runserver
```

Tente cadastrar um animal e você terá o erro:



Isso aconteceu porque você também deve informar um usuário, mas no formulário não tem ele. Você pode atualizar suas views e colocar “usuario” nos “fields”, porém esse não é o jeito certo.

É aqui que entra um novo método que iremos sobrescrever nas views: o **form_valid**

Documentação do form_valid
https://docs.djangoproject.com/en/2.2/topics/class-based-views/generic-editing/

O form_valid é o método chamado depois que submetemos um formulário, seja ele para inserir, alterar ou excluir. Ele vai verificar se os campos foram preenchidos, se se os valores estão de acordo e todas as outras verificações que se deve fazer. Por fim, ele cria um objeto e salva no banco de dados (quando chamamos o super).

Vá no “adocao/views.py” e procure o “AnimalCreate”. Ele já tem um método que sobrescrevemos, o “get_context_data”. Agora vamos sobrescrever o form_valid:

```
def form_valid(self, form):

        return super().form_valid(form)
```

Agora vamos pegar o usuário que está "logado" e inserir ele como "usuario" nesse animal
Precisamos pegar o usuário antes de chamar o super(), se não ele vai dar aquele mesmo erro. Então, adicione `form.instance.usuario = self.request.user` antes dele:

```
def form_valid(self, form):

    # Define o usuário como usuário logado
    form.instance.usuario = self.request.user

    url = super().form_valid(form)

    return url
```

O "form.instance" vai manipular uma instância desse formulário com todos os dados recebidos antes de salvar no banco. Por isso, pegamos o ".usuario" porque é o nome do atributo criado na classe. O usuário logado sempre vai estar dentro do "request.user". Aqui precisamos acessar ele pelo "self" porque estamos dentro de uma classe e queremos o usuário que fez essa requisição de salvar um animal.

A instrução retorna o endereço para onde redirecionar o usuário.

Fazer coisas depois de salvar o objeto no banco
É possível fazer coisas depois de salvar o objeto no banco. No exemplo anterior, adicionamos um usuário antes de chamar o super(). Basta fazer antes do return:

```
def form_valid(self, form):

    # Define o usuário como usuário logado
    form.instance.usuario = self.request.user

    url = super().form_valid(form)

    # código a fazer depois de salvar objeto no banco
    self.object.atributo = "algo"

    # Salva o objeto novamente
    self.object.save()

    return url
```

É possível acessar o objeto que foi criado pelo self.object. O método save() agora existe para salvar as alterações que você fez, por exemplo.

Como listar somente as Classes que são do "User"

Vamos alterar a view "AnimalList" dentro de "adocao/views.py" para listar somente os objetos que o próprio usuário que está logado cadastrou. No momento, nossa view é simples:

```
class AnimalList(LoginRequiredMixin, ListView):
    model = Animal
    template_name = "adocao/listas/list_animal.html"
```

As views que herdam a classe "ListView" tem um método que é chamado para listar todos os objetos daquela classe no banco. Nós podemos sobrescrever ele para colocar uma condição diferente do listar todos.

Método para alterar a listagem padrão de objetos

Crie o método "get_queryset" e altere o "object_list":

```
def get_queryset(self):
    # O object_list armazena uma lista de objetos de um ListView
    self.object_list =
Animal.objects.filter(usuario=self.request.user)
    return self.object_list
```

Pode ver alguma de suas listas como tem um for lá para o "object_list".

Permitir que somente o usuário dono do objeto altere ou exclua o objeto
Uma maneira de garantir que o usuário não vai alterar o ID da sua URL para tentar ver outro registro é alterando o método padrão da sua classe que herdou um DeleteView ou UpdateView. Esse método é o "get_object". É bem parecido com o da lista, porém ele recebe um ID pela URL para buscar exatamente um objeto e não uma lista deles.

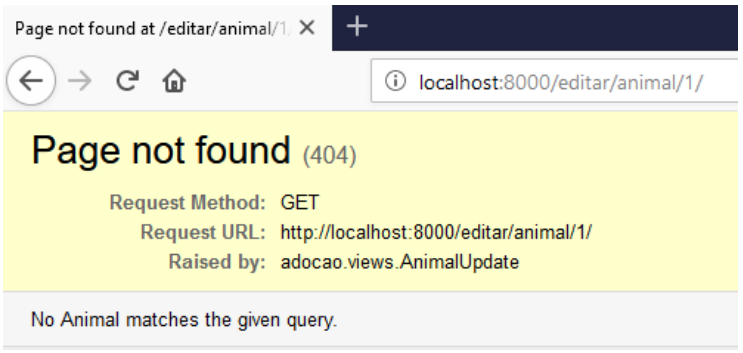
Vamos fazer um procedimento que vai resultar uma tela de erro 404 se o usuário tentar burlar nosso site. No "adocao/views.py" faça a importação:

```
# Método que busca um objeto. Se não existir, da um erro 404
from django.shortcuts import get_object_or_404
```

Crie o método a seguir no "AnimalUpdate":

```
# Altera a query para buscar o objeto do usuário logado
def get_object(self, queryset=None):
    self.object = get_object_or_404(Animal, pk=self.kwargs['pk'],
usuario=self.request.user)
    return self.object
```

O método "get_object_or_404" precisa de, pelo menos, dois parâmetros: a classe em que será feita a busca e os atributos que ela vai procurar. Neste caso procuramos pela chave primária (pk) e pelo usuário logado. Tente acessar o cadastro de algum animal com outro usuário e terá esse erro.



pk e **usuario** são atributos de **Animal**

self.kwargs['pk'] pega a chave primária da URL. Esse "**pk**" vem lá da URL:

path('editar/animal/<int:**pk**>/', ...)

Excluir somente objetos do usuário logado
O procedimento é exatamente o mesmo. Basta adicionar o método a seguir no seu "AnimalDelete":

```
# Altera a query para buscar o objeto do usuário logado
def get_object(self, queryset=None):
    self.object = get_object_or_404(Animal, pk=self.kwargs['pk'],
usuario=self.request.user)
    return self.object
```

Fazer consultas personalizadas - Documentação
https://docs.djangoproject.com/pt-br/2.2/ref/models/querysets/

É possível fazer consultas personalizadas no banco de dados. Para mais detalhes, veja a documentação acima.

Algumas consultas interessantes

Listar todos os objetos: Classe.objects.all()

Contar todos os objetos: Classe.objects.all().count()

Buscar um objeto: Classe.objects.get(pk=5)

Fazer uma consulta que retorna um ou mais objetos: Classe.objects.filter(estado="PR")

Consulta tudo que contém "Rafa": Classe.objects.get(nome__**icontains**="Rafa")

Criar um objeto: Classe.objects.create(atributo1="Rafael", atr2="Bla bla", atr3=objeto)

Tudo isso é útil para se usar no "get_context_data" e enviar dados a mais para um template

Listar todos os animais na página inicial
No “get_context_data” do “PaginaInicialView” dentro do adocao/views.py crie uma nova entrada que vai listar os últimos 10 animais cadastrados:

```
context['ultimos_animais'] = Animal.objects.all().reverse()[:10]
```

Atualize o template para listar os objetos
No seu index.html

```
{% for animal in ultimos_animais %}

<div class="float-left m-2 p-3 animais-list">
  <img width="150" height="150" src="{{animal.foto}}" alt="">
  <div class="animal-desc">
    {{animal.tipo.descricao}}
    </div>
    <span>{{animal.raca.descricao}}</span>
</div>

{% empty %}
<span>Nenhum animal cadastrado.</span>
{% endfor %}
```

Como verificar o grupo do usuário no template
É possível ver todos os grupos do usuário no template.Você consegue “pegar” o grupo como objeto pelo comando:
`request.user.groups.all.0`

Para ver se o nome é “Administrador” é necessário pegar o “name” dele. Por exemplo:

```
{% if request.user.groups.all.0.name == "Administrador" %}
<h3>Olá Admin</h3>
{% else %}
<h3>Olá usuário comum</h3>
{% endif %}
```