Elias Santos

Aula: Formulário e armazenamento

Nesse exemplo, criaremos um formulário e armazenaremos as informações digitadas
Mas antes vamos aos passos:

1) Vamos utilizar o app que já está criado como contato

2) Acrescentei no arquivo base.html mais um tag article com um bloco exclusivo para o formulário:

```
<article class="container bg-secondary">
    {% block formulario %}{% endblock formulario %}
</article>
```

3) Modificando o arquivo contato.html, incluindo agora um form:

```
{% extends 'global/base.html' %}

{% block formulario %}

<div class="container">
    <h1>Cadastrar Contato</h1>
    <form action="." method="post">
        {% csrf_token %} <!-- para evitar ataques -->
        <label for="nome">Nome</label>
        <input type="text" class="form-control" name="nome">
        <label for="nome">Idade</label>
        <input type="text" class="form-control" name="idade">
        <label for="email">Email</label>
        <input type="text" class="form-control" name="email">
        <br>
        <button class="btn btn-primary">Enviar</button>
    </form>
</div>
</body>
</html>


{% endblock formulario %}
```

Elias Santos
Aula: Formulário e armazenamento

---

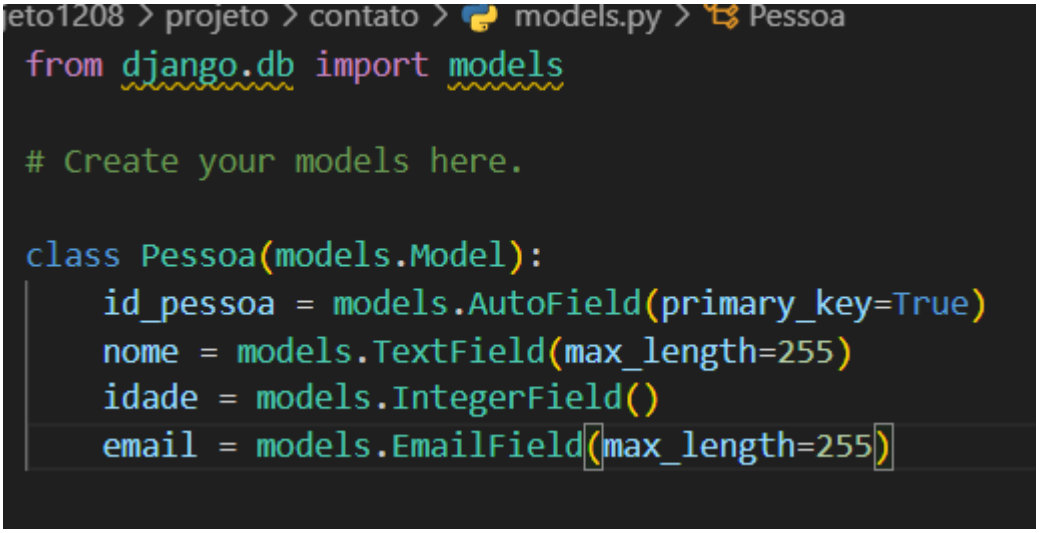
4) Criar um modelo do banco de dados para inserir os dados que estão sendo enviados pelo formulário.

Models → Definição:

No Django, o `models` é um componente fundamental do sistema de ORM (Object-Relational Mapping) que permite definir a estrutura das tabelas do banco de dados usando classes Python. Em outras palavras, o `models` no Django é responsável por mapear as classes Python para tabelas no banco de dados, permitindo que você defina a estrutura dos dados da sua aplicação de forma orientada a objetos.

Em vez de escrever SQL diretamente para criar e gerenciar tabelas, o Django oferece uma abstração mais amigável para trabalhar com bancos de dados. Você define suas classes de modelo no arquivo `models.py` do seu aplicativo Django e o ORM do Django cuida da criação, manipulação e consulta das tabelas do banco de dados correspondentes.

→ O arquivo que estamos manipulando é o arquivo **models.py** do app **contato**



Criamos uma classe denominada Pessoa, sendo inseridos 4 atributos nesta classe.
Os tipos de dados podem ser consultados ao final do documento.

Dois comandos a serem executados após o desenvolvimento de seu arquivo models.py:

a) python manage.py makemigrations
b) python manage.py migrate

→ Toda alteração que for feita em seu models, precisa ter a execução novamente destes comandos.

Uma explicação de resumo, estes comandos estão criando do modelo desenvolvido, a estrutura de seu banco de dados.
Explicação ao final deste tutorial

6) Implementando a views.py de contato:

```python
from django.shortcuts import render

#importando a classe Pessoa
from .models import Pessoa

# Create your views here.
def contato(request):
    contexto = {
        'title' : 'Contato | By Elias',
    }
    return render (
        request,
        'contato/contato.html',
        contexto
    )

def gravar(request):
    # Salvar os dados da tela para o banco
    nova_pessoa = Pessoa()
    nova_pessoa.nome  = request.POST.get('nome')
    nova_pessoa.idade = request.POST.get('idade')
    nova_pessoa.email = request.POST.get('email')
    nova_pessoa.save()

    return contato(request)
```

linha 4 importamos a classe Pessoa de models

depois foi definido a função gravar, instanciando a classe pessoa, e recebendo os dados que foram preenchidos no formulário
a instrução:
nova_pessoa.save() está salvando o registro na tabela

return contato(request) chama novamente o formulário em branco.

7) Atualizando o urls.py de contato:

projeto1208 > projeto > contato > urls.py > ...

```python
from django.urls import path
from . import views
urlpatterns = [
    path('', views.contato, name='contato'),
    path('gravar/', views.gravar, name='gravar'),
]

```

inserindo uma url dinâmica para carregar a view gravar

Elias Santos

Aula: Formulário e armazenamento

8) Chamando a url gravar no form action de contato.html

```
<h1>Cadastrar Contato</h1>
<form action="{% url 'gravar' %}" method="post">
    {% csrf_token %} <!-- para evitar ataques -->
    <label for="nome">Nome</label>
```

→ Rode a aplicação, e você já deve estar gravando os dados.
→ Para consultar se os dados estão salvos, vamos primeiro no prompt do projeto com os comandos:

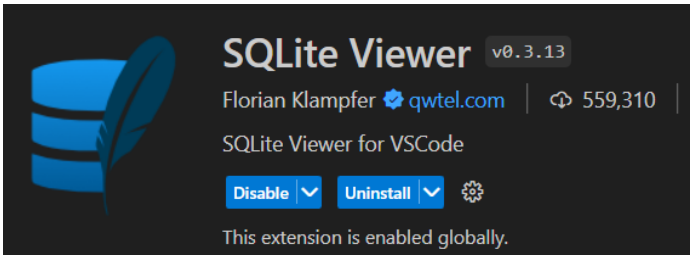
```
python manage.py shell
```

```
>>> from contato.models import Pessoa
>>> Pessoa.objects.all()
>>> pessoa = Pessoa.objects.get(id_usuario=1)
>>> print("Nome:",pessoa.nome)
```

ou para imprimir todos os campos

```
>>> print(pessoa.__dict__)
```

9) Biblioteca para visualizar o banco de dados



Com esta biblioteca instalada você não precisa mais dos comandos acima, basta acessar o arquivo do banco, o db.sqlite3

10) Criar um visualizador da tabela no projeto.

a) Vamos criar uma função na view para visualizar:

```
def exibe(request):

    # Exibir todos as pessoas
    exibe_pessoas = {
        'pessoas': Pessoa.objects.all()
    }
    # Retornar os dados para a página
    return render(
        request,
        'contato/mostrar.html',
        exibe_pessoas,


    )
```

b) fazer a url dinâmica para esta view:

```python
from django.urls import path
from . import views
urlpatterns = [
    path('', views.contato, name='contato'),
    path('gravar/', views.gravar, name='gravar'),
    path('mostrar/', views.exibe,name='mostrar'),
]
```

c) Criar o arquivo mostrar.html em contato:

```
{% extends 'global/base.html' %}

{% block formulario %}

<div class="container">
    <table class="table table-bordered table-striped">
        <tr>
            <th>id</th>
            <th>nome</th>
            <th>idade</th>
            <th>email</th>
        </tr>
        {% for pessoa in pessoas %}
        <tr>
            <td>{{pessoa.id_pessoa}}</td>
            <td>{{pessoa.nome}}</td>
            <td>{{pessoa.idade}}</td>
            <td>{{pessoa.email}}</td>
        </tr>
        {% endfor %}
    </table>

</div>


{% endblock formulario %}
```

→ Pode executar o projeto
→ Implemente para o menu ter a opção em contato de cadastro e de mostrar
→ Você também pode programar a saída de seu gravar para ele ir para a view exibe

Elias Santos

Aula: Formulário e armazenamento

Para completar o CRUD - (Create, Read, Update, Delete) faltam as opções de editar e apagar.
Vamos a elas:

11) Preparando o mostrar.html para exibir as opções de editar e apagar.
    a) Veja as alterações no arquivo:

projeto1208 > projeto > contato > templates > contato > dj mostrar.html

```
{% extends 'global/base.html' %}

{% block formulario %}

<div class="container">
    <table class="table table-bordered table-striped">
        <tr>
            <th>Id</th>
            <th>Nome</th>
            <th>Idade</th>
            <th>Email</th>
            <th>Operações</th>
        </tr>
        {% for pessoa in pessoas %}
        <tr>
            <td>{{pessoa.id_pessoa}}</td>
            <td>{{pessoa.nome}}</td>
            <td>{{pessoa.idade}}</td>
            <td>{{pessoa.email}}</td>
            <td>
                <a href={% url 'editar' pessoa.id_pessoa %}>Editar</a>
                <a href={% url 'apagar' pessoa.id_pessoa %}>Apagar</a>
            </td>
        </tr>
        {% endfor %}
    </table>

</div>


{% endblock formulario %}
```

Entrou duas opções de urls (editar e apagar) passando junto o id_pessoa do registro.

b) Estas urls dinâmicas precisam estar no arquivo urls.py:

rojeto1208 > projeto > contato > urls.py > ...

```python
from django.urls import path
from . import views
urlpatterns = [
    path('', views.contato, name='contato'),
    path('gravar/', views.gravar, name='gravar'),
    path('mostrar/', views.exibe,name='mostrar'),
    path('editar/<int:id>', views.editar, name='editar'),
    path('atualizar/<int:id>', views.atualizar, name='atualizar'),
    path('apagar/<int:id>', views.apagar, name='apagar'),

]
```

adicionamos editar, atualizar e apagar

c) Agora, a criação de suas devidas views:

```python
def editar(request, id):
    pessoa = Pessoa.objects.get(id_pessoa=id)
    return render(
        request,
        'contato/editar.html',
        {"pessoa": pessoa}
    )

def atualizar(request, id):
    pessoa = Pessoa.objects.get(id_pessoa=id)
    pessoa.nome  = request.POST.get('nome')
    pessoa.idade = request.POST.get('idade')
    pessoa.email = request.POST.get('email')
    pessoa.save()

    return exibe(request)

def apagar(request, id):
    pessoa = Pessoa.objects.get(id_pessoa=id)
    pessoa.delete()

    return exibe(request)
```

d) Por fim, a criação do arquivo editar.html, que tem a mesma estrutura do contato.html:

ojeto1208 > projeto > contato > templates > contato > dj editar.html

```
{% extends 'global/base.html' %}

{% block formulario %}

<div class="container">
    <h1>Editar Contato</h1>
    <form action="{% url 'atualizar' pessoa.id_pessoa %}" method="post">
        {% csrf_token %} <!-- para evitar ataques -->
        <label for="nome">Nome</label>
        <input type="text" class="form-control" name="nome" value="{{pessoa.nome}}">
        <label for="nome">Idade</label>
        <input type="text" class="form-control" name="idade" value="{{pessoa.idade}}">
        <label for="email">Email</label>
        <input type="text" class="form-control" name="email" value="{{pessoa.email}}">
        <br>
        <button class="btn btn-primary">Enviar</button>
    </form>
</div>
</body>
</html>

{% endblock formulario %}
```

→ Para ficar um layout interessante, pode se mudar o layout dos links de editar e apagar, usando a classe btn do bootstrap

Elias Santos

Aula: Formulário e armazenamento

---

12) Crítica da entrada das informações:

Ajuste a ser feito no views de contato → neste modelo, estamos fazendo o ajuste somente em gravar:

```python

def gravar(request):
    nome_error  = ""
    idade_error = ""
    email_error = ""
    if request.method == 'POST':
        nome   = request.POST.get('nome')
        idade  = request.POST.get('idade')
        email  = request.POST.get('email')
        if not nome: nome_error = 'O campo nome é obrigatório. '
        if not idade: idade_error = 'O campo idade é obrigatório. '
        else:
            try:
                idade = int(idade)
                if idade <= 0: idade_error = 'A idade deve ser um valor positivo. '
            except ValueError:
                idade_error = 'A idade deve ser um valor numérico. '

        if not email: email_error = 'O campo email é obrigatório. '

        if nome_error or idade_error or email_error:
            contexto = {
                'nome_error' : nome_error,
                'idade_error': idade_error,
                'email_error': email_error,
                'error' : 'Inserir valores válidos'
            }
            return render(
                request,
                'contato/contato.html',
                contexto,
            )

        # Salvar os dados da tela para o banco
        nova_pessoa = Pessoa()
        nova_pessoa.nome  = nome
        nova_pessoa.idade = idade
```

Elias Santos

Aula: Formulário e armazenamento

Ajuste a ser feito no contato.html:

```
{% extends 'global/base.html' %}

{% block formulario %}

<div class="container">
    <h1>Cadastrar Contato</h1>
    <form action="{% url 'gravar' %}" method="post">
        {% csrf_token %} <!-- para evitar ataques -->
        <label for="nome">Nome</label>
        <input type="text" class="form-control {% if nome_error %}is-invalid{% endif %}" name="nome" value={{nome}}>
        <label for="nome">Idade</label>
        <input type="text" class="form-control {% if idade_error %}is-invalid{% endif %}" name="idade" value={{idade}}>
        <label for="email">Email</label>
        <input type="text" class="form-control {% if email_error %}is-invalid{% endif %}" name="email" value={{email}}>
        {% if error %}
          {% block error %}<div class="alert alert-danger"> {{error}} </div>  {% endblock error %}
        {% endif %}
        <br>
        <button class="btn btn-primary">Enviar</button>
    </form>
</div>
</body>
</html>


{% endblock formulario %}
```

Elias Santos

Aula: Formulário e armazenamento

Complemento: Tipos de Dados

Tipos de dados de modelo básico e lista de campos

NOME DO CAMPO	DESCRIÇÃO
AutoField	É um IntegerField que aumenta automaticamente.
BigAutoField	É um número inteiro de 64 bits, muito parecido com um AutoField, exceto que é garantido para ajustar números de 1 a 9223372036854775807.
BigIntegerField	É um número inteiro de 64 bits, muito parecido com um IntegerField, exceto que é garantido para ajustar números de -9223372036854775808 a 9223372036854775807.
BinaryField	Um campo para armazenar dados binários brutos.
BooleanField	Um campo verdadeiro / falso. O widget de formulário padrão para este campo é um CheckboxInput.
CharField	É uma data, representada em Python por uma instância datetime.date.
DateField	Uma data, representada em Python por uma instância datetime.date
	É usado para data e hora, representado em Python por uma instância datetime.datetime.
DecimalField	É um número decimal de precisão fixa, representado em Python por uma instância decimal.
DurationField	Um campo para armazenar períodos de tempo.
EmailField	É um CharField que verifica se o valor é um endereço de e-mail válido.
FileField	É um campo de upload de arquivo.
FloatField	É um número de ponto flutuante representado em Python por uma instância float.
ImageField	Ele herda todos os atributos e métodos de FileField, mas também valida se o objeto carregado é uma imagem válida.
IntegerField	É um campo inteiro. Valores de -2147483648 a 2147483647 são seguros em todos os bancos de dados suportados pelo Django.
GenericIPAddressField	Um endereço IPv4 ou IPv6, em formato de string (por exemplo, 192.0.2.30 ou 2a02: 42fe :: 4).

NullBooleanField	Como um BooleanField, mas permite NULL como uma das opções.
PositiveIntegerField	Como um IntegerField, mas deve ser positivo ou zero (0).
PositiveSmallIntegerField	Como um PositiveIntegerField, mas só permite valores sob um determinado ponto (dependente do banco de dados).
SlugField	Lesma é um termo jornalístico. Um slug é um rótulo curto para algo, contendo apenas letras, números, sublinhados ou hifens. Eles geralmente são usados em URLs.
SmallIntegerField	É como um IntegerField, mas só permite valores em um determinado ponto (dependente do banco de dados).
Campo de texto	Um grande campo de texto. O widget de formulário padrão para este campo é Textarea.
TimeField	Uma hora, representada em Python por uma instância datetime.time.
URLField	Um CharField para um URL, validado por URLValidator.
UUIDField	Um campo para armazenar identificadores exclusivos universalmente. Usa a classe UUID do Python. Quando usado no PostgreSQL, ele armazena em um tipo de dados uuid, caso contrário, em um char (32).

Campos de Relacionamento
Django também define um conjunto de campos que representam relações.

NOME DO CAMPO	DESCRIÇÃO
ForeignKey	Um relacionamento muitos para um. Requer dois argumentos posicionais: a classe à qual o modelo está relacionado e a opção on_delete.
ManyToManyField	Um relacionamento de muitos para muitos. Requer um argumento posicional: a classe à qual o modelo está relacionado, que funciona exatamente da mesma forma que para ForeignKey, incluindo relacionamentos recursivos e preguiçosos.
OneToOneField	Um relacionamento um para um. Conceitualmente, é semelhante a uma ForeignKey com unique = True, mas o lado “reverso” da relação retornará diretamente um único objeto.

Elias Santos

Aula: Formulário e armazenamento

Complemento: Makemigrations e Migrate

Explicação → By Chat GPT 🙂

O comando `makemigrations` é uma funcionalidade essencial do Django que é usada para criar arquivos de migração com base nas alterações feitas nos modelos do seu aplicativo. O Django usa esses arquivos de migração para rastrear as alterações no esquema do banco de dados e permitir que você mantenha seu banco de dados em sincronia com a estrutura definida nos seus modelos.

Quando você cria ou modifica um modelo no arquivo `models.py`, o Django não altera automaticamente o esquema do banco de dados correspondente. Em vez disso, ele gera um arquivo de migração que contém as instruções necessárias para aplicar essas alterações ao banco de dados. O arquivo de migração contém comandos SQL ou equivalentes em Python para criar, modificar ou excluir tabelas, colunas, índices e outras partes do esquema do banco de dados.

O fluxo de trabalho típico ao usar o `makemigrations` é o seguinte:

> Você faz alterações nos modelos no arquivo `models.py`, como adicionar, remover ou alterar campos.
> Você executa `makemigrations` para que o Django analise suas alterações nos modelos e gere arquivos de migração apropriados.
> Você executa o comando `migrate` para aplicar as migrações pendentes ao banco de dados. Isso efetivamente sincroniza o esquema do banco de dados com as mudanças feitas nos modelos.

O `makemigrations` é uma parte fundamental do processo de desenvolvimento do Django, pois facilita a evolução do esquema do banco de dados conforme seus modelos mudam ao longo do tempo, mantendo a consistência e a integridade dos dados.