



Python para Web - Django Framework

ORM e Banco de Dados

Professor Vanderson Bossi

e-mail: Vanderson.bossi@impacta.edu.br



Django - Banco de Dados

- Todo *framework* possui suas maneiras de conectar a alguns bancos de dados.
- Conectamos ao BD - Banco de Dados - para salvar (ou persistir) os nossos dados das aplicações.
- Para conectar aos BD's, as linguagens de programação utilizam *drivers* de conexão.
- Ao conectar no BD, temos API's genéricas de acesso, leitura e escrita no BD.



- No mundo da orientação a objetos, é comum fazermos uma ponte, ou tradução, do modelo relacional para o modelo de objetos.
- Para auxiliar nessa transição, os *frameworks* costumam dispor de ferramentas auxiliares. Essas ferramentas são chamadas de ORM - Object Relational Mapping, ou Mapeamento Objeto-Relacional.
- Essas ferramentas utilizam as melhores práticas tanto de modelagem relacional quanto orientação a objetos para criar um mapa entre os dois modelos.



- Em geral temos dois mapeamentos possíveis:
 - **Mapeamento Tabelas - Classes:**
 - A ferramenta escaneia todas as tabelas criadas e cria classes para todas as tabelas primárias (não relacionamento), os relacionamentos em chave estrangeira são representados por composição de objetos.
 - **Mapeamento Classes - Tabelas:**
 - Mapeia as suas classes criadas e constrói tabelas para cada uma. Composição de objetos são tratadas como relacionamentos em chave estrangeira.



- Características de uma ferramenta ORM:
 - Mapeamento de duas mãos (classe - tabela).
 - Gerenciamento automático de conexões com o BD.
 - Identificação em classes que serão gerenciadas pela ORM (annotations ou metadados).
 - Geração automática de SQL.
 - Utilização de chave artificial numérica é preferível (em algumas é obrigatória).



- Ferramentas ORM conhecidas:
 - **Hibernate (JAVA - JPA)**
 - **EclipseLink (JAVA - JPA)**
 - **Django ORM (Python - Django)**
 - **CodeIgniter (PHP)**
 - **SQLAlchemy (Python)**
 - **ActiveRecord (Ruby)**
 - **NHibernate (.NET)**
 - **Muitos outros:**
https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software#.NET



- Como funciona no Django?
- A ORM do Django já vem instalada nativamente. Utiliza o conceito de **migrations** (migrações).
- Migrações são arquivos de diferenças entre dois estados do seu modelo (tabelas ou classes), ou seja, sempre que você alterar as suas classes é possível gerar um arquivo de diferenças no SQL (e vice-versa).
- Como conectar no BD para testar?



Django - Conectando no BD

- No arquivo **settings.py** alteramos as configurações de conexão em banco de dados. Procuramos o trecho com a seguinte lista:

```
# Database
# https://docs.djangoproject.com/en/1.11/ref/settings/#databases
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

- O Django vem por padrão com o SQLite (v3). É um micro BD com persistência e consistência de dados, perfeito para testes e aplicações aninhadas (*embedded*).

Django - Conectando no BD

- O Django oferece suporte aos BD's:
 - PostgreSQL
 - SQLite v3
 - MySQL (Python 2)
 - Oracle
- Através de bibliotecas instaladas via **pip**, mantidas por outras empresas, temos suporte também a:
 - Sybase ASE
 - SQLServer
 - MySQL (Python 3)
 - Firebird
 - IBM DB2
- Não são suportados **NoSql**.



Django - Conectando no BD

- Conectando no PostgreSQL: (**pip install psycopg2**)

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql_psycopg2',  
        'NAME': 'mydb',  
        'USER': 'myuser',  
        'PASSWORD': 'mypassword',  
        'HOST': '127.0.0.1',  
        'PORT': ''  
    }  
}
```

- Conectando no MySQL: (**pip install mysqlclient**)

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.mysql',  
        'NAME': 'mydb',  
        'USER': 'myuser',  
        'PASSWORD': 'mypassword',  
        'HOST': '127.0.0.1',  
        'PORT': ''  
    }  
}
```

Django - Conectando no BD

- Conectando no SQLServer (2005, 2008/2008R2, 2012, 2014, 2016 and Azure SQL Database): **(pip install pyodbc-azure)**

```
DATABASES = {  
    'default': {  
        'ENGINE': 'sql_server.pyodbc',  
        'NAME': 'mydb',  
        'USER': 'myuser',  
        'PASSWORD': 'mypassword',  
        'HOST': 'servidor'  
    }  
}
```



Django - Conectando no BD

- Vamos continuar usando o SQLite nos exemplos.
- Para explorar o conteúdo das tabelas do SQLite, podemos usar o SQLite Studio, uma ferramenta open source e completamente portátil (não precisa instalar nada).
- As modificações feitas via Django são independentes do SGBD que utilizarmos.
- Vamos começar a mexer no BD com o Django.



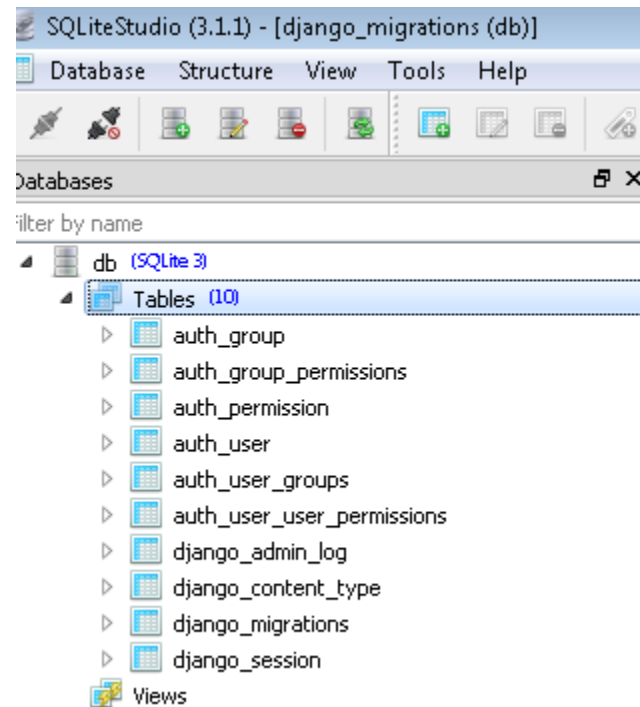
Django - BD - Migrações

- Por enquanto não temos nenhuma classe nossa para migrar para o Banco de Dados. Mas e todas aquelas aplicações já instaladas no Django?
- Elas possuem classes e modelos próprios, portanto podemos migrá-las para o BD.
- Execute o seguinte comando **python manage.py migrate** e veja o resultado





Django - BD - Migrações



```
C:\Windows\system32\cmd.exe

(venv) C:\Users\5894272\Documents\git\lmcommerce>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying sessions.0001_initial... OK

(venv) C:\Users\5894272\Documents\git\lmcommerce>_
```

Vemos que o Django já aplicou uma série de migrações vindas de várias aplicações padrões (auth, admin, sessions).



- E nossas próprias classes, como funciona?
- Para estudar melhor a modelagem de objetos, vamos começar uma nova aplicação no nosso projeto: **o catálogo de produtos.**
- Para isso vamos fazer o comando **python manage.py startapp catalogo** (sem acentos mesmo).
- Não esqueça de registrar a aplicação no **settings.py**
- Quando a aplicação estiver criada, vamos entrar no arquivo de modelos da aplicação.



Django - Primeiros Modelos

- O arquivo por enquanto está com apenas um **import**

```
from django.db import models  
  
# Create your models here.
```

- Esse **import** é importante pois todo modelo que é gerenciado pelo Django deve herdar da classe ***models.Model***
- Dentro desse arquivo vamos definir os nosso modelos do Django.
- Como estamos fazendo o nosso catálogo de vendas, vamos criar dois modelos: Categoria e Produto.



- Pequena revisão de classes em Python v3:
 - Uma classe é definida pela palavra chave **class**.
 - O nome da classe em **CamelCase**.
 - Dentro do parênteses colocamos a qual classe essa estende, se não há nenhuma, colocamos uma referência a **object**.
 - Atributos de classe são definidos sem tipos (dinamicamente tipados). Não existe as palavras chave **public** ou **private**.
 - Construtor da classe é o método **__init__**
 - Todo método recebe como primeiro parâmetro **self**, que é uma referência ao próprio objeto (similar ao **this**).
 - Não existem interfaces.
 - Podemos tentar imitar um comportamento “privado” colocando na frente de um atributo ou método o identificados **__** (duplo sublinhado).



Django - Classes Python

```
class ClasseDeObjeto (object):  
  
    atributo1 = 'valor1'  
    atributo2 = 2  
  
    def __init__(self,a1,a2):  
        self.atributo1 = a1  
        self.atributo2 = a2  
  
    def metodo1(self):  
        #faz alguma coisa  
        pass  
  
    def metodo2(self,param1,param2):  
        #faz alguma coisa  
        return param1  
  
    def __str__(self):  
        return atributo1
```



Django - Field Types

- Como o Python é dinamicamente tipado, não teríamos como definir os tipos dos atributos para as tabelas.
- Para isso, o Django possui diversos *Model Field Types* (Tipos de campos de modelo).
- Esses *Field Types* fazem com que o modelo consiga traduzir um atributo para uma coluna do SQL.
- Existem quase 30 tipos definidos.



- Alguns dos tipos comuns de SQL:
 - **CharField**: Referente ao VARCHAR do SQL
 - **DateField e DateTimeField**: Referente ao DATE e DATETIME do SQL
 - **DecimalField**: Referente ao DECIMAL do SQL
 - **IntegerField e BigIntegerField**: Referente ao INT e BIGINT do SQL
 - **TextField**: Referente ao TEXT do SQL
 - Todos os campos podem ser vistos em:
<https://docs.djangoproject.com/en/1.11/ref/models/fields/>



- Alguns tipos úteis
 - **SlugField:** Usa o VARCHAR do SQL. Usado para identificadores de texto (não usa caracteres não ASCII - acentos).
 - **AutoField e BigAutoField:** Usa os campos INT e BIGINT do SQL. São campos de auto incremento, utilizados para identificação de elementos (PK).
 - **EmailField:** Usa o VARCHAR do SQL. Valida e-mails.
 - **URLField:** Usa o VARCHAR do SQL. Valida URL's.
 - **ForeignKey:** Faz referência a outro modelo (vamos ver adiante).
 - Todos os campos podem ser vistos em:
<https://docs.djangoproject.com/en/1.11/ref/models/fields/>



Django - Primeiro Modelo

- Vamos construir nosso primeiro modelo, Categoria. Ele deve ter os atributos nome e etiqueta (um identificador especial). Fica assim:

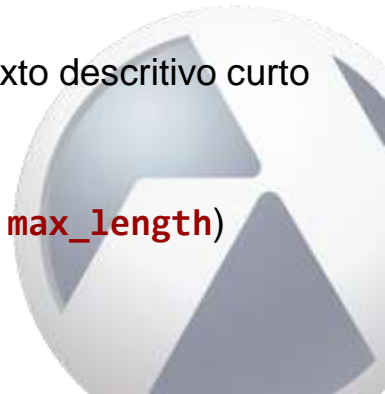
```
from django.db import models

class Categoria(models.Model):

    nome = models.CharField("Nome", max_length=50)
    etiqueta = models.SlugField("Etiqueta", max_length=50)

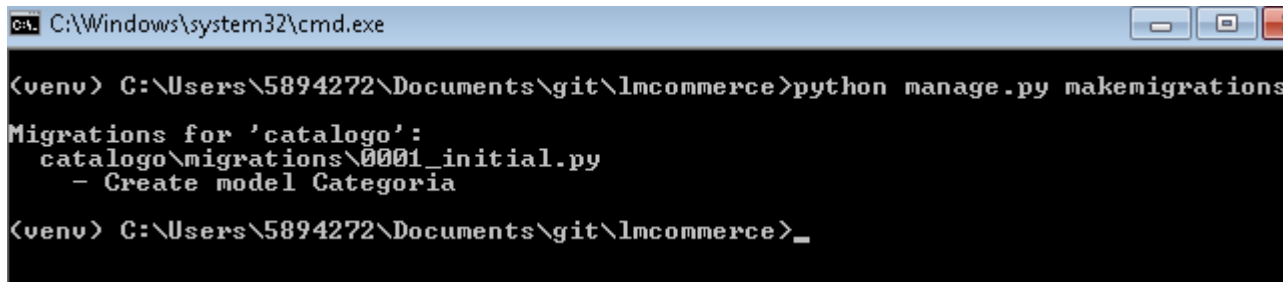
    def __str__(self):
        return self.nome
```

- No nosso primeiro modelo criamos uma classe com nome e etiqueta e a função de representação (`__str__`).
- Todo *FieldType* possui como primeiro parâmetro o **verbose_name**, que seria um texto descritivo curto sobre o atributo.
- Depois vem uma sequência de parâmetros nomeados para configurar o campo (ex: **max_length**)



Django - Primeiro Modelo

- Ao criar um primeiro modelo, vamos criar a sua migração para o BD.
- Rode o comando **python manage.py makemigrations**



```
C:\Windows\system32\cmd.exe

(venv) C:\Users\5894272\Documents\git\lmcommerce>python manage.py makemigrations

Migrations for 'catalogo':
  catalogo\migrations\0001_initial.py
    - Create model Categoria

(venv) C:\Users\5894272\Documents\git\lmcommerce>_
```

- Dentro das pastas **migrations** das aplicações que contiverem modelos novos ou alterados, vão aparecer alguns arquivos gerados.



Django - Primeiro Modelo

```
# -*- coding: utf-8 -*-
# Generated by Django 1.11 on 2017-04-25 11:43
from __future__ import unicode_literals

from django.db import migrations, models

class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Categoria',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('nome', models.CharField(max_length=50, verbose_name='Nome')),
                ('etiqueta', models.CharField(max_length=50, verbose_name='Etiqueta')),
            ],
        ),
    ]
```

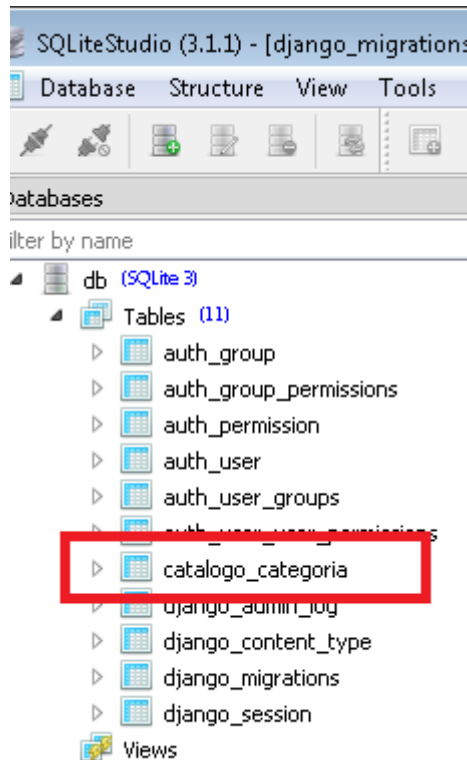

Django - Primeiro Modelo

- Esses arquivos vão conter as diferenças encontradas nos modelos e no BD, toda vez que alteramos o modelo de alguma maneira e quisermos replicar isso no BD, devemos rodar o comando **python manage.py makemigrations** para ver se tudo está certo.
- Caso a classe contenha algum erro de interpretação, durante o comando vamos saber quais erros existem.
- Se tudo estiver certo, podemos rodar o **python manage.py migrate** e olhas novamente o SQLite Studio.





Django - Primeiro Modelo



```
C:\Windows\system32\cmd.exe

<venv> C:\Users\5894272\Documents\git\lmcommerce>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, catalogo, contenttypes, sessions
Running migrations:
  Applying catalogo.0001_initial... OK

<venv> C:\Users\5894272\Documents\git\lmcommerce>_
```

- Acontece a migração agora da aplicação **catalogo**.
- É criada a tabela **catalogo_categoria**, com os campos especificados no modelo.
- Automaticamente é criado um ID numérico auto incrementado (bônus do Django).



Django - Segundo Modelo

- Com a categoria criada, vamos criar o modelo Produto no catálogo. Esse modelo deve ter os atributos: nome, identificador, preço, descrição e categoria. Temos então o segundo modelo:

```
from django.db import models

class Categoria(models.Model):
    ...

class Produto(models.Model):

    nome = models.CharField("Nome", max_length=50)
    etiqueta = models.SlugField("Etiqueta", max_length=50)
    preco = models.DecimalField("Preço", decimal_places=2,
max_digits=8)
    categoria = models.ForeignKey(Categoria)

    def __str__(self):
        return self.nome
```

- Detalhando os campos usados:
 - **CharField:** Ele configura um campo de texto para esse atributo e um VARCHAR do tamanho especificado por *max_length* (padrão ilimitado). Ele automaticamente valida tamanho de Strings
 - **SlugField:** É a mesma coisa que o CharField, mas permite apenas o uso de letras, números, sublinhado e hífen. Usado com um tipo de etiqueta (URL). No BD é criado um índice para essa coluna.
 - **DecimalField:** Configura um campo numérico para o atributo. Usa o tipo DECIMAL no BD. Tem como configuração quantos dígitos ele permite usar (*max_digits*) e, destes, quantos são usados nas casas decimais (*decimal_places*)
 - **ForeignKey:** Cria uma relação muitos-para-um (*many-to-one*) entre dois modelos. Em OO, o dono da relação que fica com a referência (o um da relação, nesse caso o Produto). O parâmetro obrigatório é uma referência ao outro modelo da relação.



Django - Segundo Modelo

- Execute os dois comandos (**makemigrations** e **migrate**) e veja o resultado

The screenshot displays the SQLiteStudio (3.1.1) interface for the 'catalogo_produto (db)' database. The left pane shows the database structure with tables listed under 'db (SQLite 3)'. The right pane shows the 'Structure' tab for the 'catalogo_produto' table, displaying its columns and their properties.

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Default value
1	id	integer	Yes				Yes		NULL
2	nome	varchar (50)					Yes		NULL
3	etiqueta	varchar (50)					Yes		NULL
4	preco	decimal					Yes		NULL
5	categoria_id	integer		Yes			Yes		NULL

Below the table structure, a terminal window shows the execution of Django migration commands:

```
(venv) C:\Users\5894272\Documents\git\lncommerce>python manage.py makemigrations
Migrations for 'catalogo':
  catalogo\migrations\0003_produto.py
  - Create model Produto

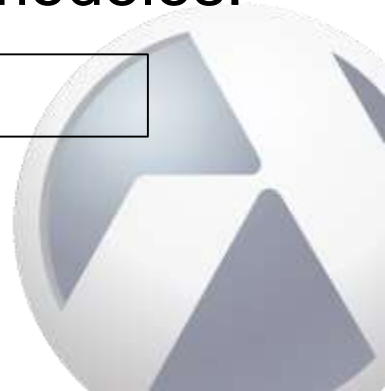
(venv) C:\Users\5894272\Documents\git\lncommerce>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, catalogo, contenttypes, sessions
Running migrations:
  Applying catalogo.0003_produto... OK

(venv) C:\Users\5894272\Documents\git\lncommerce>_
```

Django - Manipulação BD

- E como podemos manipular esses objetos no BD?
- Para testar se o acesso está correto e eficiente, temos um comando presente: **python manage.py shell**.
- Isso abre um *shell* do Python com as informações e configurações do Django já carregadas.
- A primeira coisa a fazer é importar os nossos modelos:

```
>>> from catalogo.models import Produto, Categoria
```



- Agora temos acesso aos nossos modelos. Podemos digitar o nome deles que o Python já reconhece os seus tipos:

```
>>> Categoria  
<class 'catalogo.models.Categoria'>  
>>> Produto  
<class 'catalogo.models.Produto'>
```

- Para cada modelo que estende o objeto *models.Model*, o Python adiciona uma série de ferramentas embaixo do objeto **objects**. Ao mandar imprimir o **objects** do modelo, temos a resposta:

```
>>> Categoria.objects  
<django.db.models.manager.Manager object at 0x00000000043834A8>
```

- Esse objeto **Manager** vai ser nossa interface com o BD.

- Vamos analisar os métodos mais gerais do **Manager**:
 - `all()`: Lista todas as entradas desse tipo.
 - `create(...)`: Cria uma nova entrada desse tipo.
 - `get(...)`: Obtém uma única entrada baseada em um critério (deve voltar um).
 - `filter(...)`: Lista todas as entrada que obedecem algum critério



- Criando um novo elemento:

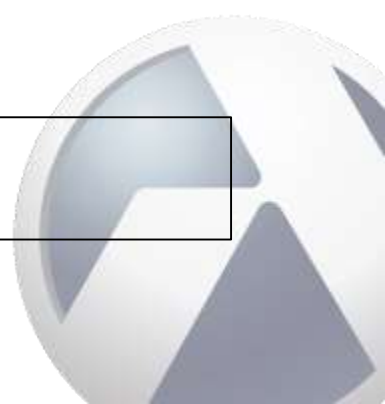
```
>>> Categoria.objects.create(nome="Python",etiqueta="python")  
<Categoria: Python>
```

- Podemos listar as entradas agora:

```
>>> categorias = Categoria.objects.all()  
>>> categorias  
<QuerySet [<Categoria: Python>]>  
>>> categorias[0]  
<Categoria: Python>
```

- No caso do **all()**, é retornado um objeto do tipo *QuerySet*, que contém uma lista dos objetos retornados do BD.
- Para usar o **get**, podemos usar:

```
>>> Categoria.objects.get(nome="Python")  
<Categoria: Python>
```



- Vamos adicionar mais uma categoria e filtrar elas:

```
>>> Categoria.objects.create(nome="Java",etiqueta="java")
<Categoria: Java>
>>> Categoria.objects.filter(nome="Python")
<QuerySet [<Categoria: Python>]>
>>> Categoria.objects.filter(nome__icontains="th")
<QuerySet [<Categoria: Python>]>
```

- Fizemos duas filtragens:
 - Por nome exato, usando **name="valor"**. Ele atua como se fosse uma cláusula WHERE name="valor" no SQL.
 - Por parte de nome. Toda vez que em filtragens usarmos o marcador **__** (dois sublinhados) após o nome de uma propriedade ele usa o **lookup**, que é uma forma de buscar por parte de nomes, nesse caso todo nome que contenha a String "th". Isso fica similar ao operador **nome LIKE "%th%"**.

- Por mais prático que seja utilizar o **shell** para testes, não é assim que vamos fazer na aplicação web.
Teoricamente deveríamos construir formulários e listagens para fazer essas operações.
- Teoricamente? Sim, pois o Django possui uma maneira automatizada de montar um CRUD (tela que cuida do cadastro e listagem de objetos).
- Lembra do <http://localhost:8000/admin> ?
- Vamos usá-lo!



Django - Criando o Admin

- Antes de usar o Django Admin, precisamos criar um usuário para ele, conhecido como **Super Usuário**.
- Para isso, rode o comando **python manage.py createsuperuser**.
- Ele irá pedir um usuário, email e senha, escolha os dois e dê enter (um bom exemplo de teste: admin com senha admin!).
- Por padrão, o Django vai exigir vários critérios para a sua senha (8 caracteres, que não seja parecido com o e-mail, etc..), para desativar isso por enquanto, altere o **settings.py**



Django - Django Admin

- Comente a seguinte região e depois tente inserir o usuário **admin** com senha **admin**:

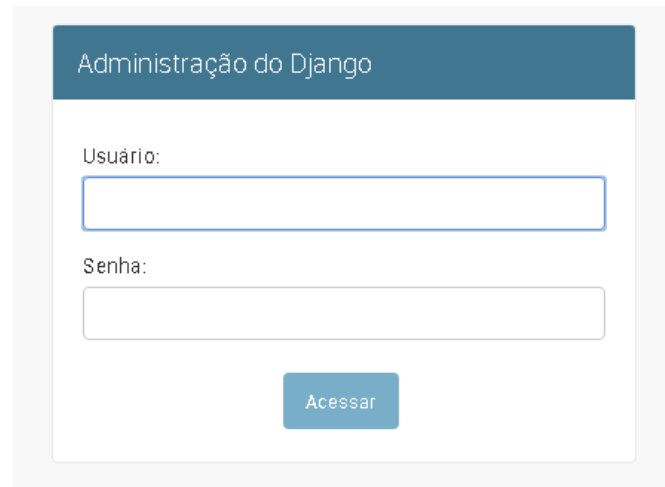
```
# Password validation
# https://docs.djangoproject.com/en/1.11/ref/settings/#auth-password-validators

AUTH_PASSWORD_VALIDATORS = [
    # {
    #     'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    # },
    # {
    #     'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    # },
    # {
    #     'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    # },
    # {
    #     'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    # },
]
```

- Agora deve ter adicionado. Rode o **runserver** e acesse a página <http://localhost:8080/admin>

Django - Django Admin

- Logue com o recém criado **admin**:



Administração do Django

Usuário:

Senha:

Acessar

Administração do Django

Administração do Site

AUTENTICAÇÃO E AUTORIZAÇÃO

Grupos

+ Adicionar

✎ Modificar

Usuários

+ Adicionar

✎ Modificar

Ações recentes

Minhas Ações

Nenhum disponível

Django - Django Admin

- Mas onde estão as listagens de produtos e categorias?
- Para que um modelo apareça no Django Admin, isso precisa ser configurado no arquivo **admin.py** em cada aplicação.
- Vamos fazer a seguinte alteração no **admin.py** da aplicação catálogo:

```
from django.contrib import admin

from catalogo.models import Produto, Categoria

admin.site.register(Categoria)
admin.site.register(Produto)
```

Django - Django Admin

- Recarregue a página do admin:

Administração do Django

Administração do Site

AUTENTICAÇÃO E AUTORIZAÇÃO

Grupos

+ Adicionar ✎ Modificar

Usuários

+ Adicionar ✎ Modificar

CATALOGO

Categorias

+ Adicionar ✎ Modificar

Produtos

+ Adicionar ✎ Modificar

Ações recentes

Minhas Ações

Nenhum disponível

- Explore um pouco a página, faça alguns cadastros. Veremos como customizar essa página para as nossas necessidades.

- Usando os comandos **makemigrations** e **migrate** do **manage.py** conseguimos criar tabelas a partir dos modelos do Python.
- Mas e se já existisse um banco de dados pronto?
- Podemos mapear de modo reverso ao que fizemos antes, ou seja, ler tabelas e gerar modelos Python a partir delas.
- Para isso temos o comando **inspectdb**.



- Ao rodar esse comando, o Django escaneia o BD inteiro em busca de todas as tabelas.
- Para cada tabela é criado um modelo Python com todos os tipos definidos.
- Todos os modelos são criados em único lugar/arquivo, sem nenhum descritivo ou *verbose_name*, uma vez que o Django não consegue adivinhar o que a coluna *primeiro_nome* pode significar.





```
$ python manage.py inspectdb
# um monte de comentários...
from django.db import models
class AuthGroup(models.Model):
    id = models.IntegerField(primary_key=True) # AutoField?
    name = models.CharField(unique=True, max_length=80)

    class Meta:
        managed = False
        db_table = 'auth_group'

class AuthGroupPermissions(models.Model):
    id = models.IntegerField(primary_key=True) # AutoField?
    group = models.ForeignKey(AuthGroup, models.DO_NOTHING)
    permission = models.ForeignKey('AuthPermission', models.DO_NOTHING)

    class Meta:
        managed = False
        db_table = 'auth_group_permissions'
        unique_together = (('group', 'permission'),)
```



- O comando coloca todo o resultado no prompt/shell, para salvar em arquivo basta redirecionar o output:

```
$ python manage.py inspectdb > models.py
```

- Depois, basta alterar os modelos gerados, colocando eles nas aplicações necessárias e utilizá-los da maneira convencional.
- Crie os *verbose_names* dos campos e remova a linha **managed=False**, dessa forma o Django consegue gerenciar esses novos modelos.



- Ao carregar os modelos no **admin**, registrando eles nos arquivos **admin.py** da aplicação, a visualização desses objetos está com problemas de acentuação em algumas palavras e a tabela mostra apenas o nome dos objetos (**__str__**):

CATALOG		
Categories	+ Adicionar	 Modificar
Products	+ Adicionar	 Modificar

Selecione category para modificar

Ação:  0 de 2 selecionados

☐ CATEGORY

☐ Design

☐ Python

2 categorys



- Para alterar essas configurações de exibição podemos adicionar aos nossos modelos os seus **metadados**.
- Em computação, tratamos **metadados** como um conjunto de dados que explica e detalha o objeto que eles referenciam.
- Por exemplo, em uma página HTML podemos usar as **metatags**, que contêm detalhes adicionais sobre a página em si (codificação, autor, compatibilidade, etc.) que definem a própria página.



- Para adicionar **metadados** para os seus modelos, dentro de cada modelo vamos definir uma **classe Meta** com as propriedades de **metadados**:

```
class Category (models.Model):  
  
    name = models.CharField("Nome", max_length=100)  
    slug = models.SlugField("Identificador", max_length=100)  
  
    def __str__(self):  
        return self.name  
  
    class Meta:  
        verbose_name = "Categoria"  
        verbose_name_plural = "Categorias"  
        ordering = ["name"]
```

- Parâmetros interessantes da classe **Meta**:
 - **verbose_name**: Um nome amigável para humanos desse objeto.
 - **verbose_name_plural**: O plural do nome amigável para humanos.
 - **ordering**: Lista de campos do objeto para serem utilizados na ordenação padrão.
 - **db_table**: Nome da tabela que esse modelo representa, caso ele tenha sido mapeado de uma tabela.



Ao salvar os **metadados** da classe **Category** temos o resultado:

Catalog administração

CATALOG		
Categorias	+ Adicionar	✎ Modificar
Products	+ Adicionar	✎ Modificar

Fazendo as mesmas alterações na classe **Product**, vamos criar a classe de **metadados** com os mesmos campos.





Django - Model Meta

```
class Product (models.Model):  
  
    name = models.CharField("Nome", max_length=100)  
    slug = models.SlugField("Identificador", max_length=100)  
    category = models.ForeignKey("catalog.Category",  
verbose_name="Categoria")  
    description = models.TextField("Descrição", blank=True)  
    price = models.DecimalField("Preço", decimal_places=2, max_digits=10)  
  
    def __str__(self):  
        return self.name  
  
    class Meta:  
        verbose_name = "Produto"  
        verbose_name_plural = "Produtos"  
        ordering = ["name"]
```

Catalog administração

CATALOG		
Categorias	+ Adicionar	✎ Modificar
Produtos	+ Adicionar	✎ Modificar

- E quanto a listagem? Ela está com um único campo, vamos modificar para colocar mais campos e em uma ordem específica.
- Para modificar a listagem de um modelo no admin do Django, precisamos modificar o modo de registro desses modelos no **admin.py**.
- Para auxiliar o Django possui a classe **ModelAdmin** que foca na configuração do *display* de um modelo na página de administração.



- Para utilizar essa classe, vamos criar uma classe para cada modelo que estenda a **ModelAdmin**.
- Com a classe criada, vamos passá-la como parâmetro ao registrar um modelo no site admin.
- Com isso o Django saberá o que mostrar na área administrativa de cada modelo registrado.



```
from django.contrib import admin
from .models import Product, Category

class CategoryAdmin(admin.ModelAdmin):

    list_display = ["name", "slug"]
    search_fields = ["name", "slug"]

class ProductAdmin(admin.ModelAdmin):

    list_display = ["name", "slug", "category"]
    search_fields = ["name", "slug", "category__name"]
    list_filter = ["category__name"]

admin.site.register(Category, CategoryAdmin)
admin.site.register(Product, ProductAdmin)
```

- Alguns parâmetros do **ModelAdmin**:
 - **list_display**: lista de campos a serem mostrados na listagem do modelo.
 - **search_fields**: campos do modelo a serem utilizados na busca desse modelo.
 - **actions**: lista de ações disponíveis (ex: remover). É possível incluir novas ações.
 - **exclude**: Lista de campos do modelo para **não** aparecerem nos formulários.
 - **list_filters**: Lista de campos para serem filtrados e um utilitário de filtragem
 - <https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#modeladmin-options>

Django - Admin

Selecione Categoria para modificar

ADICIONAR CATEGORIA +

Q Pesquisar

Ação: Ir 0 de 2 selecionados

<input type="checkbox"/>	NOME	IDENTIFICADOR
<input type="checkbox"/>	Design	design
<input type="checkbox"/>	Python	python

2 Categorias

Selecione Produto para modificar

ADICIONAR PRODUTO +

Q Pesquisar

Ação: Ir 0 de 4 selecionados

<input type="checkbox"/>	NOME	IDENTIFICADOR	CATEGORIA
<input type="checkbox"/>	Introdução ao Django Framework	introducao-ao-django-framework	Python
<input type="checkbox"/>	Introdução ao Python	introducao-ao-python	Python
<input type="checkbox"/>	Photoshop	photoshop	Design
<input type="checkbox"/>	UX Design	ux-design	Design

4 Produtos

FILTRO

Por Nome

Todos

Design

Python

- Com isso falta apenas configurar a aplicação em si.
- Para configurar os **metadados** de cada aplicação, vamos usar o arquivo **apps.py**.
- Novamente vamos criar uma classe que conterá os **metadados**. Essa classe deve estender a **AppConfig** já importada no **apps.py**



- A **apps.py** fica:

```
from django.apps import AppConfig

class CatalogConfig(AppConfig):
    name = "catalog"
    verbose_name = "Catálogo"
```

- Para funcionar, devemos registrar essa configuração como padrão no módulo Python. Vamos alterar o **__init__.py** da aplicação:

```
default_app_config = "catalog.apps.CatalogConfig"
```

- Com isso o resultado fica:

Administração do Site

AUTENTICAÇÃO E AUTORIZAÇÃO		
Grupos	+ Adicionar	✎ Modificar
Usuários	+ Adicionar	✎ Modificar
CATÁLOGO		
Categorias	+ Adicionar	✎ Modificar
Produtos	+ Adicionar	✎ Modificar



- A classe **AppConfig** possui os seguintes parâmetros de configuração:
 - **name:** O caminho completo para a aplicação (usando convenção do Python). Ex: `django.contrib.auth`
 - **label:** Nome curto da aplicação. Ex: `auth`
 - **verbose_name:** Nome amigável para humanos da aplicação. Ex: `Autenticação`
 - **path:** Caminho absoluto para a aplicação.



- Com essas poucas configurações já conseguimos dar uma cara mais customizada ao admin do Django.
- É possível alterar o *layout* do site estendendo os templates que geram o admin.
- Também existem diversos pacotes que cuidam dessas extensões para deixar o site mais responsivo ou para adicionar novos itens na tela.
- <https://djangopackages.org/grids/g/admin-interface/>



- Agora que temos uma maneira de cadastrar os produtos e suas categorias via interface web, precisamos lista-los para os visitantes da loja.
- Podemos simplesmente criar uma nova **view**, chamando o método *Product.objects.all()* e devolver eles para o template renderizar.
- Mas já deve ser possível perceber que vários dessas **views** tem um comportamento muito parecido: devolvem um template específico, lista todos os modelos de um tipo, lista os modelos de um tipo com um critério, etc.

Django - Classes Chamáveis

- Como as **views** são métodos, fica difícil reaproveitar o funcionamento de cada uma delas de uma maneira orientada a objetos.
- O Python possui um artifício para essas dificuldades que são as **classes ‘chamáveis’**, ou *callable classes*.
- Basicamente, através de uma configuração da classe, podemos usar uma instância de objeto como se fosse uma função:

```
peessoa = Pessoa()  
peessoa() # isso executa algo de fato.
```



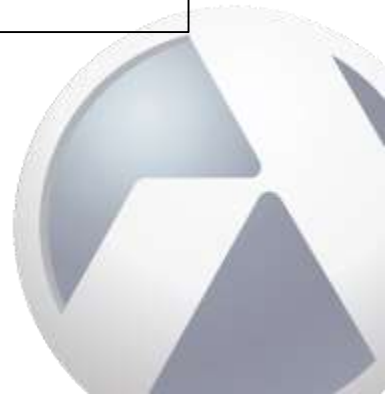
Django - Classes Chamáveis

- Para criar uma **classe chamável**, a sua classe deve implementar o método `__call__`:

```
class Pessoa(object):  
  
    def __init__(self, nome, idade):  
        self.nome = nome  
        self.idade = idade  
  
    def __call__(self):  
        print(self.nome, self.idade)  
  
p = Pessoa("Yuri", 30)  
p.nome  
p.idade  
p()
```

- O resultado desse chamado seria:

```
"Yuri"  
30  
"Yuri 30"
```



Django - Classes Chamáveis

- Ao implementar o método `__call__`, todas as instâncias podem ser chamadas, ou executadas, passando parâmetros quaisquer nessa execução.
- E se criássemos uma classe chamável para uma view, ex:

```
class IndexView(object):  
  
    def __call__(self, request):  
        return render(request, "index.html")
```

- Para que isso funcione como a função anterior (`def index(request)`), precisamos de uma **instância** com o nome apropriado. Como o nome da função era **index** vamos criar a instância: **index = IndexView()**
- Substitua a função pela classe e veja a mudança.



Django - Class Based Views

- O nome dessas classes no django é ***Class Based Views***, onde temos nossas views definidas por classes ao invés de funções.
- O poder das **Class Based Views (CBV's)** está na orientação objeto. Todo reaproveitamento de código que a orientação a objetos traz, está presente nas CBVs.
- Vamos fazer a listagem de produtos pensando em CBV's.



Django - Class Based Views

- A listagem de produtos deve pegar todos os produtos cadastrados e colocar no contexto, além de devolver o template de produtos. Temos então:

```
class ProdutosView(object):  
  
    def __call__(self, request):  
        context = {  
            "produtos": Produto.objects.all()  
        }  
        return render(request, "catalog/produtos.html", context)  
  
produtos = ProdutosView()
```

- Poderíamos ainda criar uma classe mais genérica que a **IndexView** (view que retorna um template) ou a **ProdutosView** (view que lista um modelo), usando a orientação a objetos.



Django - Class Based Views

- Pensando nisso, o Django já possui diversas CBV's implementadas para os tipos mais comuns.
- Todas as CBV's do Django herdam de **django.views.generic.View**. Essa **view** genérica define apenas alguns métodos auxiliares para a utilização das CBV's.
- Ao invés de implementarmos o **__call__** do Python, seria melhor implementar algum método que faça mais sentido para a web, portanto, classes que herdem de **django.views.generic.View** devem implementar os métodos **get** e **post** (conforme a necessidade).
- Também, como não vamos usar o **__call__**, ao invés de executar a instância, vamos chamar o método auxiliar **as_view()**.

Django - Class Based Views

- A **IndexView** e a **ProdutosView** ficam assim então:

```
from django.views.generic import View
class IndexView(View):
    def get(self, request):
        return render(request, "index.html")
class ProdutosView(View):
    def get(self, request):
        context = {
            "produtos": Produto.objects.all()
        }
        return render(request, "catalog/produtos.html", context)
index = IndexView.as_view()
produtos = ProdutosView.as_view()
```

- Já temos CBV's com métodos que fazem mais sentido para o mundo web, mas não podemos fazer lógicas de views mais genéricas (retornar templates ou listar modelos)?



Django - Class Based Views

- Dentro do **django.views** existem várias **views** genéricas prontas para uso.
- O site <http://ccbv.co.uk> possui uma descrição detalhada e exemplos dessas CBV's genéricas.
- As mais básicas que vamos utilizar são:
 - **TemplateView**: view que simplesmente devolve algum template.
 - **ListView**: view de listagem de modelos (com ou sem filtragem)
 - **FormView**: view de formulário de modelo (define o get e o post)
 - **DetailView**: view para mostrar detalhes de um produto específico.

Django - Class Based Views

- Vamos alterar o **IndexView** para usar o **TemplateView**:

```
from django.views.generic import View, TemplateView

class IndexView(TemplateView):
    template_name = "index.html"

index = IndexView.as_view() # pode ficar no urls.py também
```

- Para a **TemplateView**, basta passar o parâmetro **template_name** com o caminho do template a ser renderizado.

```
from django.views.generic import View, ListView

class ProdutosView(ListView):
    model = Produto
    template_name = "catalogo/produtos.html"
    context_object_name = "produtos"

produtos = ProdutosView.as_view()
```

Django - Pendências de Listagem

- Para fechar as listagens, temos duas pendências:
 - Listagem do menu em **todas as views**: o menu é um item que deve ser listado em todos os lugares do sistema, ao invés de simplesmente copiar a listagem para todas as views que devolvem templates, podemos fazer isso de uma maneira genérica?
 - Filtrar produtos por **categoria**: ou filtrar objetos por parâmetros da URL. Queremos ser capazes de passar parâmetros para os nossos **gets**, sendo possível, por exemplo, listar produtos por categoria.

Django - Context Processors

- Se quisermos que exista uma listagem presentes em **todas** as requisições, temos um conjunto de objetos que executam sempre antes delas, são os **Context Processors**.
- No **settings.py** temos alguns **context processors** instalados por padrão:

```
TEMPLATES = [  
    {  
        ...  
        'OPTIONS': {  
            'context_processors': [  
                'django.template.context_processors.debug',  
                'django.template.context_processors.request',  
                'django.contrib.auth.context_processors.auth',  
                'django.contrib.messages.context_processors.messages',  
            ],  
        },  
    },  
]
```



Django - Context Processors

- Esses processadores executam funções específicas antes de todas as requisições (verifica sessão, processa requisição, constrói mensagens, etc.)
- Para criar o seu próprio processador de contexto, crie a classe na aplicação correspondente (nesse caso a **core**)

```
# arquivo core.context_processor.py
from catalog.models import Category

def categories(request):
    return {
        "categories" : Category.objects.all()
    }
```

- Todo método dos processadores de contexto devem receber a **requisição** e retornar um **contexto** para ser utilizada no resto da requisição.



Django - Context Processors

- Para usar o seu processador, basta agora registrá-lo no **settings.py**:

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [],  
        'APP_DIRS': True,  
        'OPTIONS': {  
            'context_processors': [  
                'django.template.context_processors.debug',  
                'django.template.context_processors.request',  
                'django.contrib.auth.context_processors.auth',  
                'django.contrib.messages.context_processors.messages',  
                "core.context_processors.categories"  
            ],  
        },  
    ],  
]
```

- Pronto, suas categorias estão sendo listadas em todas as requisições

Django - Filtrando pela URL

- Para podermos filtrar os nossos dados pela URL, existe a maneira clássica e maneira mais moderna (utilizada no REST).
 - A maneira clássica: se quisermos listar todos os produtos por uma única categoria, nós podemos passar o id dessa categoria como uma **query parameter** na URL (ex: http://localhost:8000/catalogo/?category_id=1)
Essa maneira, por mais funcional, não é a maneira mais natural de se passar informações.
 - A maneira moderna: ao invés de passar uma **query parameter**, podemos usar um pedaço da URL como variável de consulta e, ao invés de usar um id, podemos usar um texto mais amigável (ex: slug). Ex: <http://localhost:8000/catalogo/design>

Django - Filtrando pela URL

- Para utilizar a maneira mais moderna precisamos aprender a:
 - Extrair parâmetros de pedaços da URL (**url-path**).
 - Vamos definir na **urls.py** regiões da URL que serão tratados como parâmetros do get.
 - Filtrar modelos específicos usando algum critério na **ListView**.
 - Vamos criar uma **CategoriaView** que acesse os dados retornados da URL e liste todas os produtos de uma categoria específica.



Django - Filtrando pela URL

- Queremos então que a URL <http://localhost:8000/catalogo/CATEGORIA> liste todos os produtos da categoria **CATEGORIA** (ex: design).
- Vamos definir então a **urlpatterns** da seguinte maneira:

```
url(r'^(?P<slug>[\w_-]+)/$', views.categoria)
```

- Nessa **regex**, temos algumas novas definições:
 - **(...)** marca uma região que será capturada quando a expressão interna der *match* com a URL
 - **?P<nome>** identifica o nome do parâmetro quando a expressão der *match* com a URL
 - **[...]** marcar quais tipos de caracteres devem estar presentes na URL para dar *match*
 - **\w** usa quaisquer caracteres alfanuméricos (números e letras comuns).

Django - Filtrando pela URL

- Portanto, o `urls.py` irá capturar toda url que utilizar apenas número, letras sem acento, sublinhados e hífens.
- Esse padrão é exatamente o utilizado pelas etiquetas (**slugs**)
- Agora vamos construir a view (**CategoriaView**) que vai receber e renderizar essas informações.
- Ela será praticamente igual ao **ProdutoView**, com a diferença que ela deve filtrar os produtos de acordo com a categoria. Para isso, vamos implementar o método **get_query_set**, que por padrão, na classe mãe, lista todos os modelos do tipo especificado.

Django - Filtrando pela URL

```
from django.views import generic

class CategoriaView(generic.ListView):
    template_name = "catalogo/categoria.html"
    context_object_name = "produtos"

    def get_queryset(self):
        return Produto.objects.filter(categoria__slug=self.kwargs["slug"])

categoria = CategoriaView.as_view()
```

- Para o **CategoriaView** temos que:
 - mudar o **template_name** (agora categoria.html).
 - remover o atributo **model** (não vamos listar tudo).
 - implementar o método **get_query_set** (esse método diz como buscar os modelos no lugar do `.all()`)
 - retornar a busca usando `.filter(...)`
- Dentro do `.filter(...)` apareceram um **kwargs**, o que é isso?



Django - Filtrando pela URL

- Na assinatura do método **get_query_set** não há acesso às variáveis de requisição ou dos parâmetros de url. Isso é passado dentro da instância da **view** (no parâmetro **self**).
- Entre outras coisas, o parâmetro **self** possui três atributos importantes:
 - **self.request**: referência ao objeto de requisição atual.
 - **self.args**: Lista de argumentos passados na URL sem nome (sem o ?P<nome> no urls.py).
 - **self.kwargs**: Conhecido como *known arguments* - argumentos nomeados - contém um dicionário com todos os parâmetros nomeados da URL.
- Por isso, ao acessar o **self.kwargs["slug"]** estamos pegando o valor do parâmetro passado na URL.



Django - Filtrando pela URL

- Se quisermos passar mais coisas no contexto, que não sejam só a lista de produtos, podemos sobrescrever o método **get_context_data**
- Esse método por padrão, para o **ListView**, apenas obtém o resultado do método **get_query_set** e coloca dentro do contexto com o nome definido na propriedade **context_object_name**.

```
from django.views import generic

class CategoriaView(generic.ListView):
    template_name = "catalogo/categoria.html"
    context_object_name = "produtos"

    def get_context_data(self, **kwargs):
        context = super(CategoriaView, self).get_context_data(**kwargs)
        context["categoria_atual"] = get_object_or_404(Categoria, slug=self.kwargs["slug"])
        return context

    def get_queryset(self):
        return Produto.objects.filter(categoria__slug=self.kwargs["slug"])

categoria = CategoriaView.as_view()
```

Django - Filtrando pela URL

- O que está acontecendo no **get_context_data**:
 - Primeiro executamos o comportamento padrão da função usando o **super** (passando o nome da classe e a referência para o objeto)
 - O parâmetro ****kwargs** é usado para acessar todos os argumentos nomeados da função, nesse caso, **get_context_data**
 - Pegamos a categoria representada pela etiqueta (**slug**) na URL (a função **get_object_or_404** é um utilitário que volta uma mensagem de erro caso não exista a categoria).
 - Essa categoria é colocada na variável de contexto **categoria_atual**.
- Pronto, podemos colocar quaisquer variáveis no contexto agora dessa **view**.





Django - Exercício para Fazer Junto

- Usando a mesma ideia da listagem por categoria, ao clicar no produto nós gostaríamos de mostrar uma tela de detalhes dele. Tente refazer a lógica para mostrar os detalhes de um produto
- A view genérica **DetailView** é a mais indicada para isso. Ela obtém um único membro de um modelo pela **pk**, ou por um **slug**.
- Além dos parâmetros de template e e contexto que apresentamos, podemos configurar como usar o **slug** para obter um único produto

- **Sugestão de view:**

```
class ProdutoDetalhesView(generic.DetailView):  
    model = Produto  
    # template_name: por padrão é produto_detail.html na mesma aplicação  
    # slug_field: o nome do campo slug no seu modelo (etiqueta?)  
    # slug_url_kwarg: o nome do parâmetro na URL do slug.
```





Python para Web Django Framework

Formulários HTML e Django Forms



- Para entender como incluir e alterar dados na web fazemos em geral o seguinte fluxo:
 - Usamos os formulários no HTML para enviar as informações necessárias.
 - No servidor, pegamos a requisição POST e retiramos os parâmetros necessários, que são passados pelos atributos **names** dos inputs do formulário.
 - Cada dado deve ser validado e, se houver erro, deve voltar a mensagem de erro.
 - Caso os dados estejam corretos, devemos montar o objeto a ser inserido/alterado usando um modelo pré-definido.
 - Com o objeto, precisamos usar a camada de persistência (conexão no BD) e executar a rotina SQL para inserir ou alterar dados (já feito pelo ORM do Django)

- Mesmo com pequenas alterações no funcionamento, em geral todos as inserções e alterações vão seguir esse mesmo fluxo.
- Como em outros casos, para evitar a repetição dessa mesma lógica, o Django possui uma ferramenta para automatizar esse fluxo: **django.forms.Form**
- O Django **Form** permite que se crie uma classe que represente a lógica de um formulário no HTML (inclusive a estrutura se quisermos, veremos adiante), onde a tradução e validação dos dados do POST serão feitos de maneira automática.

- Para mostrar as maneiras de utilizar o Django Form, vamos usar o formulário de contato como exemplo.
- O formulário de contato é um composto de três campos: nome, e-mail e mensagem, além de um botão para submeter.
- O formulário deve validar se tudo está preenchido, se o e-mail é válido e deve mandar um e-mail caso esteja ok, se houver algum erro, deve reportar.





- Exemplo do template “contato.html”

```
{% extends "base.html" %}

{% block title %}
    Contato | {{ block.super }}
{% endblock %}

{% block container %}
<div class="page-header">
    <h1>Contato</h1>
</div>
<form action="" method="POST" >
    <p>
        <label for="nome">Nome</label>
        <input name="nome" id="nome">
    </p>
    <p>
        <label for="email">E-Mail</label>
        <input name="email" id="email">
    </p>
    <p>
        <label for="mensagem">Mensagem</label>
        <textarea name="mensagem" id="mensagem" rows="20" cols="50"></textarea>
    </p>
    <input type="submit" value="Enviar"/>
</form>
{% endblock %}
```


- A view contato (por enquanto sem Class Based View):

```
def contato (request):  
    return render(request, "contato.html")
```

- Por enquanto, apenas retornamos o template correto (lembrar de adicionar no **urls.py**)
- Para trabalharmos com formulários no Django, precisamos de um modelo que identifique esse formulário. Vamos criar o arquivo **forms.py** na aplicação

```
from django import forms  
  
class ContactForm(forms.Form):  
    nome = forms.CharField(label="Nome", required=True)  
    email = forms.EmailField(label="E-mail", help_text="Informe um E-mail válido")  
    mensagem = forms.CharField(label="Mensagem", widget=forms.Textarea(),  
required=True)
```

- Da mesma forma que os modelos, os **forms** do Django possuem *Field Types* para definir os tipos de campos que o formulário HTML está usando.
- Com isso podemos definir em uma classe quais os campos que serão usados no POST, validar seu preenchimento, tamanho e etc.
- A configuração é similar: temos os mesmos tipos de *Field Types* que os modelos, mas ao invés de passar um *verbose_name*, usamos a opção **label**.



- Algumas opções comuns dos *FieldTypes* dos **forms**:
 - **required**: Indica se o campo é obrigatório (True or False)
 - **label**: Indica um texto para ser usado dentro do **label** do HTML (veremos um exemplo).
 - **initial**: Indica um valor inicial para o campo.
 - **widget**: Os campos utilizados pelo Django em geral são inputs do tipo text, essa opção pode passar um tipo diferente para o campo. Ver mais em <https://docs.djangoproject.com/en/1.11/ref/forms/widgets/>
 - **help_text**: Indica um texto de ajuda para esse campo.



- Para associar o **form** a view, colocamos ele como uma variável de contexto:

```
from .forms import ContatoForm

def contato (request):
    form = ContatoForm()
    context = {
        "contato.html": form
    }
    return render(request, "contato.html", context)
```

- Por enquanto ainda não interagimos com o **form**. Para isso, precisamos separar a lógica do POST e a do GET. Durante o GET, tudo que fazemos é enviar o template para o navegador.
- Apenas o POST vai fazer alguma coisa com o formulário em si.

- Quando a requisição for POST, o formulário deve primeiro validar os seus dados e depois fazer algo com os dados.
- Vamos criar no **ContatoForm** uma função de envio de e-mail (que por enquanto não faz nada) para marcar o que deve ser feito com os dados do formulário:

```
from django import forms

class ContactForm(forms.Form):
    nome = forms.CharField(label="Nome", required=True)
    email = forms.EmailField(label="E-mail", help_text="Informe um E-mail válido")
    mensagem = forms.CharField(label="Mensagem", widget=forms.Textarea(), required=True)

    def envia_email(self):
        pass
```

- Vamos então separar a lógica do POST e do GET:

```
from .forms import ContatoForm

def contato (request):
    form = ContatoForm()
    if request.POST:
        pass
    else:
        pass
    context = {
        "contato.html": form
    }
    return render(request, "contato.html", context)
```

- Dentro do if, vamos validar e, se válido, enviar o email. Para preencher os *FieldTypes* do **ContatoForm** devemos passar no construtor o objeto POST (que terá a informação mandada do formulário)
- Para validar o formulário, o Django Form possui o método `is_valid()`

- Mandando os valores para o form e testando se é valido, fica assim:

```
from .forms import ContatoForm

def contato (request):
    if request.POST:
        form = ContatoForm(request.POST)
        if form.is_valid():
            form.envia_email()
    else:
        form = ContatoForm()
    context = {
        "contato.html": form
    }
    return render(request, "contato.html", context)
```

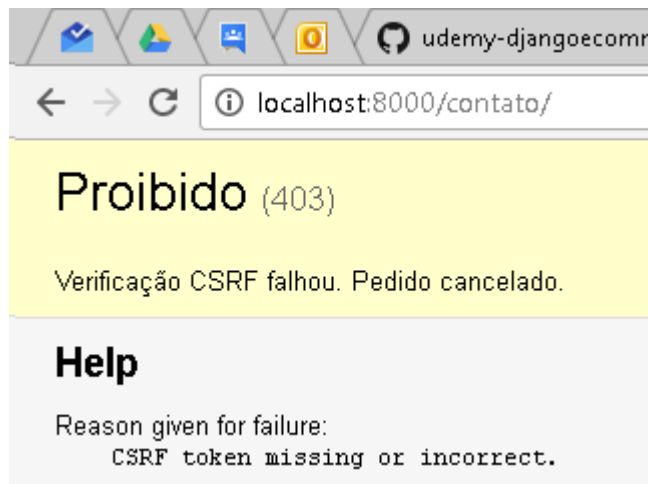
- Para testar, vamos alterar o método **envia_email** para apenas imprimir as informações.



```
def envia_email(self):  
    print(  
        "Nome:", self.cleaned_data["nome"],  
        "Email:", self.cleaned_data["email"],  
        "Mensagem:", self.cleaned_data["mensagem"]  
    )
```

Vamos usar o método **cleaned_data** para conseguir os dados dos *FieldTypes*

Tente agora submeter o formulário vazio para ver o que acontece:



- O Django te protege automaticamente do ataque *Cross-Site-Request-Forgery* (https://pt.wikipedia.org/wiki/Cross-site_request_forgery)
- Para passar, coloque dentro do **form** a template tag {% csrf_token %}

- Ao tentar novamente (com o token) a página se recarrega e nada acontece (nem no prompt), isso porque o formulário está inválido, mas nenhuma mensagem é exibida.
- Para exibir as mensagens de erro temos duas templates tags para utilizar, logo antes do **form** no HTML:
 - `{{ form.non_field_errors }}`: mostra erros no processamento do formulário
 - `{{ form.errors }}`: mostra erros nos campos do formulário.
- Inclua as duas template tags e tente novamente, vai ver que logo antes do formulário aparecerão mensagens, de uma forma bem simples

- nome
 - Este campo é obrigatório.
- email
 - Este campo é obrigatório.
- mensagem
 - Este campo é obrigatório.

- Agora envie o formulário com dados válidos e olhe o seu prompt que a mensagem foi impressa.
- E se agora quisermos adicionar o *FieldType* para digitar o assunto no **form**? Precisamos alterar o **ContatoForm** bem como o formulário HTML. Seria melhor que alterássemos apenas um lugar.
- O Django Form adiciona alguns template tags para geração de formulários, dado que eles são gerados sempre com a mesma cara. Quando passamos o formulário no parâmetro **form** no contexto, podemos alterar o formulário de contato para ser gerado dinamicamente.



Django - Form

```
{{ form.non_field_errors }}
<form action="" method="POST" >
  {% csrf_token %}
  {% for field in form %}
    <p>
      {{ field.errors }}
      {{ field.label_tag }} {{ field }}
      {% if field.help_text %}
        <p class="help">{{ field.help_text|safe }}</p>
      {% endif %}
    </p>
  {% endfor %}
  <input type="submit" value="Enviar"/>
</form>
```

Com isso, o formulário é gerado para todos os campos, com os tipos definidos no **ContatoForm**, veja que as mensagens de erro dos campos e a de ajuda ficam perto dos campos em si.



- E se, ao invés de usar o método, quiséssemos usar uma *Class Based View*?
- Para isso, temos a view genérica **FormView**, vamos trocar a **view** atual para:

```
class ContatoView(FormView):  
    template_name = 'contato.html'  
    form_class = ContatoForm  
    success_url = 'obrigado'  
  
    def form_valid(self, form):  
        form.envia_email()  
        return super(ContatoView, self).form_valid(form)  
  
contato = ContatoView.as_view()
```

- Essa view encapsula toda a lógica que implementamos antes.

- Mas e se tivéssemos um modelo, como faríamos para salvar no banco de dados.
- Imagine o modelo a seguir:

```
from django.db import models

class Contact(models.Model):

    email = models.EmailField("Email", max_length=30,
null=False)
    name = models.CharField("Nome", max_length=50, null=False)
    message = models.TextField("Mensagem")

    class Meta:
        verbose_name="Contato"
        verbose_name_plural="Contatos"
```

- Note que as informações no ContactForm e no Contact(Model) estão muito parecidas:
 - ambos têm um campo e-mail
 - ambos têm um campo nome
 - ambos tem um campo de mensagem
- Para evitar esse tipo de repetição, temos o **django.forms.ModelForm**.
- Esse formulário monta um Django Form a partir de um modelo diretamente.



- Vamos alterar o form de contato para mandar e-mail ao salvar:

```
from django import forms
from .models import Contact
class ContactForm(forms.ModelForm):

    class Meta:
        model = Contact
        fields = "__all__"

    def save(self):
        novo_contato = super(ContactForm, self).save()
        self.envia_email(contato=novo_contato)
        return novo_contato

    def envia_email(self, contato):
        message = "Nome:{0}\nE-
mail{1}\n{2}".format(contato.name, contato.email, contato.message)
        print("Contato Django Ecommerce\n", message)
```

- No lugar dos campos, usamos a classe Meta para configurar o modelo e os campos a serem usados (__all__ usa todos).

Django - ModelForm

- Estendemos o uso do método `save` para enviar o e-mail **após** a inserção.
- O método envia e-mail agora usa o modelo propriamente dito.
- Na **View** podemos trocar a chamada **`form.envia_email`** para **`form.save()`** e teremos o comportamento desejado.



- Já temos um comportamento bastante padronizado para a criação de objetos no Django:
 - Cria-se o modelo.
 - Cria-se o form que representa o modelo.
 - Cria-se uma FormView para gerenciar a inserção.
- Dá para ser mais direto? Sim! Com a **CreateView**
- a **CreateView** é uma view baseada em classe para a inserção de um modelo no banco de dados. Vamos ver o exemplo dela para o Contato.



- Já temos um comportamento bastante padronizado para a criação de objetos no Django:
 - Cria-se o modelo.
 - Cria-se o form que representa o modelo.
 - Cria-se uma FormView para gerenciar a inserção.
- Dá para ser mais direto? Sim! Com a **CreateView**
- a **CreateView** é uma view baseada em classe para a inserção de um modelo no banco de dados. Vamos ver o exemplo dela para o Contato.





Python para Web Django Framework

Autenticação e Área de Usuário



Django - Autenticação

- O Django Admin já possui uma tela de login, mas ela não deve ser usada para usuários comuns, por exemplo, clientes da sua loja.
- Para isso precisamos criar uma outra maneira de autenticar o nosso usuário (e também para sair do site ou desfazer o login).
- Apesar de haverem diversas formas de login (senha, token, etc.), todas possuem a mesma lógica:
 - usuário manda um identificador (ex: nome) e um verificador (ex: senha).
 - sistema compara com o que existe na base e volta True caso esteja correto ou False caso contrário.



Django - Autenticação

- O django já possui no módulo **django.contrib.auth** a base da implementação de autenticação que precisamos, basta alterarmos alguns detalhes.
- Uma das questões é o modelo de usuário. O modelo do Django pode não possuir tudo que precisamos para os nossos usuários. Dessa forma, seria bom estender esse modelo para conter os parâmetros que vamos precisar.
- Vamos criar uma aplicação para gerenciar o cadastro de usuário e a área do cliente: **aplicação contas**
python manage.py startapp contas



- Dentro do arquivo **models.py** devemos criar o modelo de usuário. Esse modelo deve estender o **AbstractBaseUser** do Django:

```
from django.contrib.auth.models import AbstractBaseUser, UserManager
ALUNO = 'A'
PROFESSOR = 'P'
COORDENADOR = 'C'
PERFIS = (
    (ALUNO, 'Aluno'),
    (PROFESSOR, 'Professor'),
    (COORDENADOR, 'Coordenador')
)
class Usuario(AbstractBaseUser):
    ra = models.IntegerField("RA", unique=True)
    nome = models.CharField("Nome", max_length=100, blank=True)
    email = models.EmailField("E-Mail", unique=True)
    ativo = models.BooleanField("Ativo", default=True)
    perfil = models.CharField("Perfil", max_length=1, choices=PERFIS)

    def __str__(self):
        return self.nome
```

- Para funcionar como um modelo de autenticação, precisamos configurar mais algumas coisas:
 - No modelo devemos outras configurações:
 - Uma propriedade definindo qual campo o Django usará como *username* (login)
USERNAME_FIELD = "ra"
 - O *Objects* do usuário não pode ser o comum, vamos usar um próprio para usuários
objects = UsuarioManager()
 - O modelo precisa de dois métodos para visualização: um para nome completo e outro para nome curto:
def get_full_name(self):
 return self.nome
def get_short_name(self):
 return self.nome
 - Precisamos por último indicar os campos obrigatórios:
REQUIRED_FIELDS = ["email", "nome"]



- Para usar no Django Admin, precisamos de três métodos auxiliares:

```
def has_perm(self, perm, obj=None):  
    return True  
def has_module_perms(self, app_label):  
    return True
```
- Esses métodos são para substituir o sistema padrão de permissões do Django.
- Por último, precisamos de um substituo para a propriedade que diz se o usuário em questão pode acessar o Django Admin:

@property

```
def is_staff(self):  
    return self.perfil == 'C'
```





- A classe **UsuarioManager** precisa ser modificada para aceitar o RA ao invés de username:

```
class UsuarioManager(BaseUserManager):
    use_in_migrations = True

    def _create_user(self, ra, password, **extra_fields):
        if not ra:
            raise ValueError('RA precisa ser preenchido')
        user = self.model(ra=ra, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_user(self, ra, password=None, **extra_fields):
        return self._create_user(ra, password, **extra_fields)

    def create_superuser(self, ra, password, **extra_fields):
        return self._create_user(ra, password, **extra_fields)
```

- No arquivo **settings.py** devemos adicionar:
 - Novo modelo de autenticação:
AUTH_USER_MODEL = "contas.Usuario"
 - A url de login (para redirecionamentos)
LOGIN_URL = 'login'
 - A url para onde ir quando o login der certo
LOGIN_REDIRECT_URL = 'index'
 - A url de logout
LOGOUT_URL = 'logout'
- Para funcionar agora, precisamos migrar o novo modelo para o BD. Como já temos uma tabela de usuário lá (padrão), o melhor é remover os modelos e mandar inserir novamente (com **makemigrations** e **migrate**).
- Depois de migrado o site, precisamos colocar a **view** de login e o mecanismo de **logout**.

- Para criação de novos perfis (tipos de usuário) no Django, podemos criar novos modelos que estendam o atual modelo de usuário:

```
...  
class Aluno(Usuario):  
  
    parent_link = models.OneToOneField( #Apenas para modelos pré-existent  
        Usuario,  
        primary_key=True,  
        db_column="usuario_id",  
        parent_link=True  
    )  
    curso = models.ForeignKey(  
        'curriculo.Curso',  
        blank=True,  
        null=True,  
    )
```

- Desse novo modelo, surgirá uma tabela associada a tabela de usuário.

- Para usar o Django Admin para criar novos usuários, devemos usar o uma página Admin específica:

```
from django.contrib.auth.admin import UserAdmin

class AlunoAdmin(UserAdmin):

    list_display = ('email', 'nome', 'curso')
    list_filter = ('perfil',)
    fieldsets = ( (None, {'fields': ('email', 'nome', 'curso')}),)
    add_fieldsets = ((None, { 'fields': ('ra', 'email', 'nome', 'curso')} ),)
    search_fields = ('email',)
    ordering = ('email',)
    filter_horizontal = ()

admin.site.register(Aluno, AlunoAdmin)
```



- Além disso, ao salvar um novo usuário do tipo aluno, queremos garantir duas coisas:
 - Que seja gerada uma senha provisória para o usuário, assim ele trocará ela após o primeiro login.
 - Que o perfil na tabela usuário para esse mesmo aluno seja sempre 'A' (Aluno).
- Para garantir esses requisitos, devemos sobrescrever os formulários de novo aluno e de alteração de aluno.



Django – Admin e Novo Usuário

```
from django import forms

class NovoAlunoForm(forms.ModelForm):

    class Meta:
        model = Aluno
        fields = ('ra', 'email', 'nome', 'curso')

    def save(self, commit=True):
        user = super(NovoAlunoForm, self).save(commit=False)
        user.set_password('123@mudar')
        user.perfil = 'A'
        if commit:
            user.save()
        return user

class AlterarAlunoForm(forms.ModelForm):

    class Meta:
        model = Aluno
        fields = ('email', 'nome', 'curso')

class AlunoAdmin(UserAdmin):
    form = AlterarAlunoForm
    add_form = NovoAlunoForm

...
```



Django - Login e Logout

- No arquivo **urls.py**, vamos importar as views **login** e **logout** do Django:
from django.contrib.auth.views import login, logout
- Vamos criar as **urls** 'entrar' e 'sair' para ambas as views:
`url(r"^entrar/", login, {"template_name": "login.html"}),`
`url(r"^sair/", logout, {'next_page': 'index'}),`
- Para a **view login** temos uma opção adicional de passar o template que vamos usar para o login (veremos a seguir).
- Para a **view logout** precisamos dizer para onde a **view** irá ao fazer o logout (nesse caso o index)



- Vamos criar o template **form.html** na aplicação **core**:

```
{% extends "base.html" %}
{% block title %}Entrar | {{ block.super }}{% endblock %}
{% block container %}
<div>
  <div>
    <div>
      <h1>Entrar</h1>
    </div>
    <form method="post">
      {% csrf_token %}
      {% for field in form %}
        <p>
          {{ field.errors }}
          {{ field.label_tag }} {{ field }}
        </p>
        <input type="submit" value="Entrar">
      {% endfor %}
    </form>
  </div>
</div>
{% endblock %}
```



Django - Login e Logout

- Agora crie um super usuário novamente (**python manage.py createsuperuser**) e tente logar usando o novo formulário:



- Esse erro indica que não há **view** com o nome **index**. Ao definir uma url para uma view qualquer, podemos sempre definir um nome identificador para usar no sistema, ao invés de usar a url. Por exemplo:
url(r'^\$', index, name="index")

Django - Login e Logout

- Para verificar que estamos logados, no **base.html** podemos adicionar o seguinte trecho no menu:

```
{% if user.is_authenticated %}
    <li><a href="{% url 'logout' %}">Sair</a></li>
    <li> Olá {{ user }}</li>
{% else %}
    <li><a href="{% url 'login' %}">Entrar</a></li>
{% endif %}
```

- Assim temos um menu condicional se o usuário está ou não logado.
- Note o uso da template tag {% url 'login' %}, ela converte automaticamente a URL com o nome passado (nesse caso login) no seu arquivo **urls.py**.



Django – Bloqueando Views

- Agora que já temos uma autenticação funcionando no sistema, devemos fazer uma maneira de bloquear algumas views para determinados usuários.
- O bloqueio vai acontecer em dois níveis:
 - Views que precisam que o usuário esteja logado.
 - Views que precisam que o usuário tenha um determinado perfil.
- Para executar essa ação, vamos utilizar dois **decorators** do Django
(<https://pythonhelp.wordpress.com/2013/06/09/entendend-o-os-decorators/>)

Django – Bloqueando Views

- Vamos supor duas Views: uma apenas para professores e outra apenas para alunos.
- Primeiro precisamos bloquear essas views para usuários logados.
- Para isso vamos usar o decorator **@login_required**

```
from django.contrib.auth.decorators import login_required
```

```
@login_required(login_url='/entrar')  
def aluno(request):  
    return render(request, "aluno.html")
```

```
@login_required(login_url='/entrar')  
def professor(request):  
    return render(request, "professor.html")
```

Django – Bloqueando Views

- Agora para bloquear views para usuários específicos vamos precisar de funções que verifiquem as nossas condições.
- Essas funções recebem o usuário como parâmetro de entrada e testem o que é necessário (nesse caso se o perfil está certo)

```
def checa_aluno(user):  
    return user.perfil == 'A'  
  
def checa_professor(user):  
    return user.perfil == 'P'
```

Django – Bloqueando Views

- Com essas duas funções prontas, podemos usar o decorator **@user_passes_test**

```
from django.contrib.auth.decorators import login_required, user_passes_test
# Funções de teste...
@login_required(login_url='/entrar')
@user_passes_test(checa_aluno, login_url='/?error=acesso', redirect_field_name=None)
def aluno(request):
    return render(request, "aluno.html")
@user_passes_test(checa_professor, login_url='/?error=acesso', redirect_field_name=None)
@login_required(login_url='/entrar')
def professor(request):
    return render(request, "professor.html")
```



F a c u l d a d e
IMPACTA
T E C N O L O G I A
