Unidade 10 - Integrando Django com SGBD

Apresentação

O Django é um *framework* de desenvolvimento web que utiliza a linguagem Python como base para a criação de aplicações, sejam elas completas, com *front end* e *back end*, ou apenas na camada *back end*. Em ambos os casos, manipular os objetos do banco de dados se torna uma tarefa bastante simples se comparada ao trabalho em diversos outros *frameworks* de desenvolvimento web.

Essa simplicidade se dá pela forma como o Django foi projetado, utilizando o padrão de *model-template-view*, que é similar ao *model-view-controller*, além de ferramentas de automação existentes no Django. Nesse caso, o trabalho de manipulação dos objetos e do banco de dados se dá pela camada model desse padrão, que será abordada em mais detalhes nesta Unidade de Aprendizagem.

Nesta Unidade de Aprendizagem, você vai aprender sobre como funciona o banco de dados no Django, mais especificamente sobre o sistema de gerenciamento de banco de dados SQLite, além dos arquivos de configuração do Django e como configurá-los para trabalhar com o MySQL.

Bons estudos.

Ao final desta Unidade de Aprendizagem, você deve apresentar os seguintes aprendizados:

- Reconhecer o SGBD SQLite.
- Descrever os arquivos de configuração do Django.
- Modificar a configuração para usar MySQL.

Infográfico

O Django tem a proposta de fornecer agilidade à equipe de desenvolvimento na criação de aplicações, bem como cuidados com segurança e escalabilidade. Ele utiliza padrões e conceitos como o *model-template-view* e o ORM, permitindo ao desenvolvedor criar aplicações melhores padronizadas e com menos código.

Neste Infográfico, você verá como funciona o ORM do Django a partir das classes que herdam de *model* e das migrações.

CLASSES MODELO E ORM DO DJANGO

O Django utiliza o padrão *model-template-view* e as classes *model* automatizam boa parte do processo de criação das tabelas do banco de dados da aplicação.

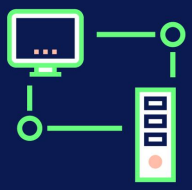
MODELO

Nas classes modelo são definidas as características e os comportamentos do objeto que se deseja armazenar pela aplicação no banco de dados. Essas classes herdam de `django.db.models.Model`. Cada classe vira uma tabela no banco de dados e cada atributo vira uma coluna da tabela. O método `__str__()` é responsável por gerar uma string amigável para o objeto instanciado quando seu conteúdo for acessado.

```
class Emplacamento(models.Model):
    TIPOS_VEICULO = [
        ('0', 'Moto'),
        ('1', 'Carro'),
        ('2', 'Van'),
    ]

    placa = models.CharField(max_length=8)
    data = models.DateField()
    tipo_veiculo = models.CharField(max_length=1, choices=TIPOS_VEICULO)

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

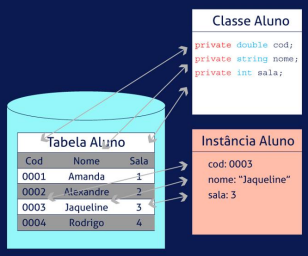


MIGRATIONS

Uma vez que os novos modelos são definidos pelos desenvolvedores, é necessário criar um arquivo de migração que funciona como um arquivo que guarda as alterações da versão do banco de dados a partir da atualização das classes modelo. **Os comandos para isso são** `python manage.py makemigrations` e `python manage.py migrate`.

MAPEAMENTO DE OBJETO RELACIONAL

O ORM pode ser um recurso que melhora a produtividade da equipe de desenvolvimento tanto na implementação de novas funcionalidades do projeto quanto em sua manutenção. Com o ORM, **o acesso ao banco de dados é abstraído, permitindo ao desenvolvedor acessar os dados diretamente com os objetos**, sem precisar escrever SQL.



MANIPULAÇÃO DE DADOS

Com o acesso ao ***shell***, o desenvolvedor pode acessar e manipular o banco de dados da aplicação de forma a realizar cadastros para realizar testes. Para isso, basta instanciar um objeto com a devida classe em uma variável, adicionar os parâmetros à variável e em seguida chamar o método ***save*** herdado da classe `models.Model`. Para consultar um objeto a partir do identificador basta chamar `Emplacamento.objects.get(id=1)`.

```
>>> e = Emplacamento()
>>> e.placa = 'AAA12E4'
>>> e.data = '2019-11-01'
>>> e.tipo_veiculo = '1'
>>> e.save()
```

CURIOSIDADES

TECNOLOGIAS ORM

Algumas das tecnologias que utilizam o conceito de ORM são o Django, o Laravel para PHP, o Hibernate para Java, o Ruby on Rails, o Sequelize, entre outras.

ACTIVE RECORD

O Django ORM é similiar ao Active Record do Rails, mas nesse caso utiliza o Python como linguagem de manipulação dos dados, ao invés do Ruby.

INTERFACE ADMIN

O Django tem um painel administrativo criado junto com a aplicação para permitir ao administrador manipular os dados via interface de sistema.

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

Conteúdo do Livro

É difícil pensar em uma aplicação que não utilize banco de dados e que não tenha a necessidade de armazenar os dados em uma base, recuperá-los sempre que solicitado pelo usuário, alterá-los quando necessário e excluí-los quando se tornam desnecessários. Dependendo do objetivo da aplicação e da forma que os dados precisam ser tratados, pode ser necessário escolher um ou outro SGBD (sistema gerenciador de banco de dados), embora existam casos nos quais mais do que um deles pode resolver o problema do desenvolvedor, mesmo que cada um tenha as suas características, tanto na forma de utilização quanto na forma de configuração no projeto.

No capítulo Integrando Django com SGBD, da obra *Programação Back End I*, você vai conhecer a respeito do SGBD SQLite, poderá observar os arquivos de configuração do Django, bem como irá descobrir como conectar a aplicação e o projeto em desenvolvimento em um banco de dados MySQL.

PROGRAMAÇÃO BACK END I

Fabiano Berlinck Neumann

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Integrando Django com SGBD

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Reconhecer o sistema gerenciador de banco de dados (SGBD) SQLite.
- Descrever os arquivos de configuração do Django.
- Modificar a configuração do Django para usar o MySQL.

Introdução

Os desenvolvimentos Web e *mobile* estão em constante evolução e seu aprendizado tem se tornado cada vez mais simples. No entanto, em algumas aplicações é necessário utilizar bancos de dados para armazenar informações.

No Django, um projeto pode utilizar as bibliotecas REST API para expor os dados armazenados no servidor para as aplicações *mobile*. Seus projetos Web *front end* podem ser desenvolvidos com bibliotecas JavaScript. Um projeto no Django pode, também, ser desenvolvido totalmente em Python e buscar os dados para apresentá-los nos formulários, que são criados automaticamente. Em ambos os casos, será necessário configurar um SGBD para armazenar e manipular os dados.

Neste capítulo, você conhecerá o SGBD SQLite, além de aprender sobre os arquivos de configuração do Django. Você também verá como se configura um projeto para utilizar o banco de dados MySQL em uma aplicação Web.

SGBD SQLite

O Django possibilita o uso de diversos sistemas de gerenciamento de banco de dados, como o SQLite e o MySQL, que serão abordados neste capítulo. Existem outros SGBD que podem ser encontrados em documentações, tutoriais e exemplos na Internet.

Devido a sua simplicidade, o SGBD SQLite é bastante utilizado em aplicações móveis para armazenar dados nos dispositivos dos usuários, tanto no desenvolvimento nativo quanto no híbrido, assim como em servidores Web criados com Django. Além disso, o SQLite é o banco padrão do Django, como pode ser observado com o arquivo `db.sqlite3` da Figura 1, que é criado automaticamente quando o projeto é executado pela primeira vez.

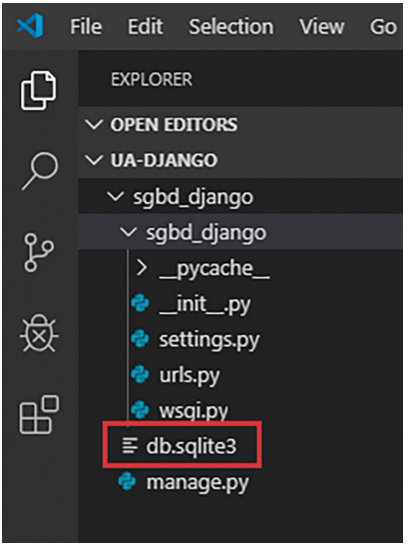


Figura 1. Arquivo `db.sqlite3`.

Considerando que o desenvolvedor já tenha realizado a configuração inicial do ambiente de desenvolvimento, como a instalação do Python, do Django e esteja no diretório em que deseja criar o projeto, digite os comandos a seguir no console para criar um projeto que tenha a mesma estrutura de pastas e arquivos da Figura 1.

```
django-admin startproject sgbd_django
cd sgbd_django
python manage.py runserver
```

Caso nenhum erro seja lançado no console, é possível abrir o navegador e acessar o endereço `http://localhost:8000` para visualizar a tela de sucesso na instalação do projeto Django, que é a tela padrão apresentada até que o desenvolvedor crie telas e configure novas URL.

Como podemos observar nos comandos, após a criação do projeto ocorre a mudança para o diretório criado, onde está o arquivo `manage.py`, utilizado para rodar o projeto pela primeira vez no servidor com o comando `runserver` do Python. Nesse momento é criado o arquivo `db.sqlite3`, no qual ficarão as tabelas e os dados da aplicação. A configuração completa do arquivo de configurações do Django, para que o banco funcione, será apresentada no segundo tópico deste capítulo.

Além dos comandos de configuração é necessário criar as classes modelo para adicionar elementos no banco de dados e realizar outros tipos de operações com eles. No Django, o padrão utilizado é o *Model-Template-View* (MTV), similar ao *Model-View-Controller* (MVC). Em ambos os padrões, a camada *Model* está intimamente relacionada ao banco e fornece os dados para as outras duas camadas.

Modelos e manipulação de dados com o SQLite3

Para definir um modelo, basta criar uma classe que herda de `models.Model`, conforme o exemplo de trecho de código a seguir, e definir os atributos e métodos da classe, que definem as características e o comportamento do modelo.

No caso de uma *sprint*, que na metodologia *scrum* refere-se a um período de tempo bem definido para executar uma lista de tarefas, o modelo (classe `Sprint`) contará com uma data inicial (`start_date`), uma data final, (`end_date`) e uma lista de tarefas, definidas como chave estrangeira na classe tarefa (`Task`), com deleção dos objetos em cascata, quando os objetos `Task` também são excluídos com a *sprint*.

```
from django.db import models

class Sprint(models.Model):
    start _ date = models.DateField()
    end _ date = models.DateField()

    def _ _ str _ _ (self):
        return self.start _ date

class Task(models.Model):
   sprint = models.ForeignKey(Sprint, on _ delete=models.CASCADE)
    name = models.CharField(max _ length=200)
    complete = models.BooleanField()

    def _ _ str _ _ (self):
        return self.name
```

Os tipos de campos do banco de dados são definidos a partir dos tipos de atributos definidos na classe `models`, como o `DateField()`; que representa um campo de data, o `ForeignKey()`; que é uma referência a outro objeto por meio de uma chave estrangeira ou ID do objeto referenciado; o `CharField()`, para campos de texto, neste caso limitado a 200 caracteres; e o `BooleanField()`, para campos com valor verdadeiro ou falso.

No trecho de código a seguir é possível observar como criar um objeto da classe `Sprint` e armazená-la no banco de dados.

```
from datetime import datetime
from nome _ do _ app.models import Task, Sprint

sprint1 = Sprint(start _ date=datetime(2019, 11, 2), end _
date=datetime(2019, 11, 9))
sprint1.save()

sprint1.create(name="Tarefa 1", complete=False)
sprint1.create(name="Tarefa 2", complete=False)
sprint1.save()
```

Para imprimir na tela o ID da *sprint* e de cada uma das tarefas, é possível utilizar os comandos a seguir.

```
print(sprint1.id)
print(sprint1.task _ set.all())
print(Sprint.objects.all())
```

O primeiro `print` apresenta apenas o identificador do objeto na tela. O segundo `print` apresenta todas as tarefas do objeto `sprint1` devido ao `task_set.all`, um padrão do Django no qual a primeira parte do `task_set` vem do nome do objeto, e a segunda, refere-se ao `set`, que significa conjunto em inglês. Já o terceiro `print` apresenta todas as *sprints* cadastradas no banco de dados.

Se for necessário obter uma ou mais tarefas com um determinado nome, é possível utilizar o comando `Sprint.objects.get(name="Tarefa 1")`, assim como é possível utilizar o `get` para obter todas as *sprints* cadastradas com uma determinada data inicial. Para excluir um objeto específico e todos os seus filhos, quando descrito em modo cascata com o `on_delete=models.CASCADE`, basta utilizar o comando `sprint1.delete()`.

Arquivos de configuração do Django

O banco de dados configurado para ser utilizado por padrão na criação de um projeto é o SQLite, conforme apresentado no tópico anterior. De acordo com Rubio (2017), esse banco de dados incluído na distribuição do Python é um banco relacional leve, que pode ser substituído por outros bancos também suportados oficialmente pelo Django, como o MySQL, o PostgreSQL e o Oracle, assim como por bancos que utilizam pacotes desenvolvidos por terceiros.

Para configurar o banco de dados no Django é necessário acessar o arquivo `settings.py` e, em seguida, alterar os valores em `DATABASES`, que inicia com um dicionário Python apresentado no trecho de código a seguir, definido em pares de chave e valor.

```
import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.
abspath(__file__)))

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Ainda para Rubio (2017), essa configuração inicial do banco com SQLite é a forma mais rápida de configuração de base do dados, já que não exige outros pacotes ou credenciais para que a conexão seja estabelecida com a base. O arquivo do banco `db.sqlite3`, criado automaticamente, é configurado como valor da chave `'NAME'`. A chave `'ENGINE'` é utilizada para mudar o SGBD para outro banco de dados e alguns de seus valores podem ser encontrados no tópico a seguir.

Assim como a chave `'default'`, outras chaves podem ser definidas dentro do mesmo bloco, referenciando outros nomes de bancos de dados em que seu valor é um novo dicionário Python com os parâmetros de conexão com o banco de dados. No caso da referência `'default'`, esta indica que todas as operações de banco de dados devem ser realizadas a partir de tais parâmetros, caso não seja especificado em qual banco deve-se manipular os dados.

Quando se adiciona uma aplicação ao projeto com o comando `python manage.py startapp core`, é necessário adicionar `'core.apps.CoreConfig'` dentro da lista `INSTALLED_APPS` do arquivo `settings.py`, conforme apresenta a Figura 2, no qual o `core` é o nome dado à aplicação deste capítulo, mas que poderia ter outro nome, como `main`. Dessa forma, o texto para adicionar ao arquivo `settings.py` seria `'main.apps.MainConfig'`.

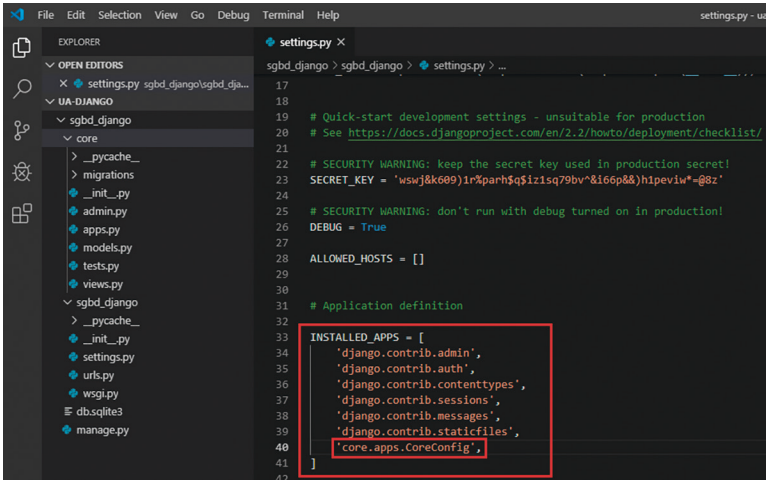


Figura 2. Adicionando a configuração de um aplicativo novo iniciado no projeto.

Após adicionar a configuração do aplicativo ao arquivo `settings.py` e definir as classes modelo que representam as tabelas do banco de dados, é necessário executar comandos de migração no terminal.

Migrations

As migrações (*migrations*) funcionam como um sistema de controle de versão de um banco de dados que precisam ser informadas ao Django quando forem realizadas modificações no banco, além de ficarem registradas para que possam ser analisadas e revertidas, se necessário.

De acordo com Battisti (2019), o Django utiliza o *object relational mapper* (ORM), ou mapeador de objeto relacional, que abstrai a criação da estrutura do banco como as tabelas a partir das classes modelo declaradas no arquivo `models.py`.

Com os comandos a seguir, os desenvolvedores informam ao Django que existem mudanças nas classes modelo e que é necessário transformar os atributos das classes em colunas de tabelas.

```
python manage.py makemigrations
python manage.py migrate
```

Com o comando `makemigrations`, o Django cria um arquivo que representa os modelos na forma de SQL. Com o comando `migrate` são aplicadas as alterações desse arquivo no banco de dados.

Como conectar o projeto com o MySQL

De acordo com Rubio (2017), o nome de referência `'default'` de `DATABASES` no arquivo `settings.py` representa que as operações de criação, leitura, atualização e deleção serão realizadas de acordo com o banco de dados definido no valor da chave `'default'`, que nesse caso são as chaves `'ENGINE'`, para representar qual será o banco, e `'NAME'`, para representar o nome da instância da base de dados. Para o autor, o padrão mais importante é o `'ENGINE'`, que apresenta diferentes possibilidades de valores que podem ser aplicados a ele, conforme pode ser observado no Quadro 1.

Quadro 1. Valores do Django para o `ENGINE` para usar diferentes bancos de dados

Banco de dados	Valor `ENGINE` do Django	Pacotes necessários
MySQL	`django.db.backends.mysql`	Incluso com o Django
Oracle	`django.db.backends.oracle`	Incluso com o Django
PostgreSQL	`django.db.backends.postgresql_psycopg2`	Incluso com o Django
SQLite	`django.db.backends.sqlite3`	Incluso com o Django
SAP (Sybase) SQL Anywhere	`sqlany_django`	`pip install sqlany-django`
IBM DB2	`ibm_db_django`	`pip install ibm_db_django`

(Continua)

(Continuação)

Quadro 1. Valores do Django para o `ENGINE` para usar diferentes bancos de dados

Banco de dados	Valor `ENGINE` do Django	Pacotes necessários
Firebird	`firebird`	`pip install django-firebird`
ADO – Microsoft SQL Server	`sqlserver_ado`	`pip install django-mssql`
ODBC – Microsoft SQL Server, Azure SQL ou outro banco de dados compatível com ODBC	`sql_server.pyodbc`	`pip install django-pyodbc-azure` ou `pip install django-pyodbc`

Fonte: Adaptado de Rubio ([201-?], documento *on-line*).

Apesar de a lógica de aplicação em Django estar associada de forma neutra ao banco de dados, o que permite ao desenvolvedor utilizar muitas vezes comandos que não dependem da seleção deste, existem diferenças em algumas operações entre bancos de dados diferentes.

Para Battisti (2019), o SQLite não é indicado para ser utilizado em aplicações que rodem no ambiente de produção. O autor sugere que o banco seja trocado por outro mais robusto, como o PostgreSQL ou o MySQL antes do `deploy` da aplicação.

Para fazer isto, basta alterar o dicionário definido em `DATABASES` no arquivo `settings.py`, configurando o arquivo conforme apresentado no trecho de código a seguir.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'sgbd _ django',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

O campo `'NAME'` deve ser configurado com o nome do banco de dados criado ao qual se pretende conectar a aplicação, bem como o usuário e a senha de acesso ao banco, além do `'HOST'` e do `'PORT'`, configurados para o banco de dados. Feito isso, caso o `mysqlclient` esteja instalado de forma adequada, basta executar os comandos `python manage.py makemigrations` e `python manage.py migrate` novamente. Assim, o desenvolvedor está pronto para criar mais classes modelo, manipular os dados e configurar outros bancos, quando necessário.

Referências

BATTISTI, M. Models e migrations no Django 2. *Hora de Codar*, [*S. l.*], 2 jan. 2019. Disponível em: https://www.horadecodar.com.br/2019/01/02/models-e-migrations-no-django-2/. Acesso em: 14 nov. 2019.

RUBIO, D. *Beginning Django*: web application development and deployment with Python. New York: Apress, 2017. 593 p.

RUBIO, D. Set up a database for a Django project. *Web Forefront*, [*S. l.*], [201-?]. Disponível em: https://www.webforefront.com/django/setupdjangodatabase.html. Acesso em: 14 nov. 2019.

Leituras recomendadas

DATABASE Functions. *Django Software Foundation*, [*S. l.*], 2019. Disponível em: https://docs.djangoproject.com/en/2.2/ref/models/database-functions/. Acesso em: 14 nov. 2019.

RAMOS, V. A. *Desenvolvimento web com Python e Django*. [*S. l.*]: Python Academy, 2018. 130 p. Disponível em: https://pythonacademy.com.br/assets/ebooks/desenvolvimento-web-com-python-e-django/desenvolvimento-web-com-python-e-django.pdf. Acesso em: 14 nov. 2019.

RAMOS, V. A. Desenvolvimento Web com Python e Django: Model. 26 maio 2018. *Python Academy*, [*S. l.*], Disponível em: https://pythonacademy.com.br/blog/desenvolvimento-web-com-python-e-django-model. Acesso em: 14 nov. 2019.

SETTING Django up to use MySQL. *Stack Overflow*, New York, 4 Oct. 2013. Disponível em: https://stackoverflow.com/questions/19189813/setting-django-up-to-use-mysql. Acesso em: 14 nov. 2019.

SQLITE3 Database Setup. *Tech with Tim*, [*S. l.*], Apr. 2019. Disponível em: https://techwithtim.net/tutorials/django/sqlite3-database/. Acesso em: 14 nov. 2019.

Fique atento

Os *links* para *sites* da Web fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Dica do Professor

Dependendo da aplicação projetada, é possível existir a necessidade
do aplicativo ter mais do que um banco de dados. Para isso, é necessário configurar cada um dos bancos no arquivo de configurações do projeto e em muitos casos informar em qual banco de dados a operação será realizada.

Nesta Dica do Professor, você verá como funciona a manipulação de múltiplos bancos de dados em uma mesma aplicação, como configurar os bancos no arquivo settings.py, como funcionam as sincronizações com os bancos de dados, além de como selecionar um banco manualmente para uma operação.

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.

Exercícios

1) **Existem comandos que automatizam parte do processo de criação dos modelos na forma de comandos SQL, em um arquivo para cada versão e aplicação das alterações deste no banco de dados. Marque o comando que cria o arquivo com as alterações das classes modelo na forma de comandos SQL.**

A) python manage.py makemigrations.

B) python manage.py migrate.

C) python makemigrations manage.py.

D) python migrate manage.py.

E) python manage.py runserver.

2) **Quando o desenvolvedor cria o projeto e a aplicação em seu ambiente de desenvolvimento com Django, ele pode executar um comando para rodar um servidor localmente e testar no ambiente de desenvolvimento à medida que cria a solução. Marque a opção que representa o comando que inicia o servidor.**

A) django-admin startproject nomeprojeto.

B) python manage.py runserver.

C) from django.db import models.

D) python manage.py migrate.

E) python manage.py makemigrations.

3) **O Django utiliza o conceito de ORM (*object relational mapper*), ou mapeador de objeto relacional, que abstrai a criação das tabelas do banco a partir dos atributos descritos nas classes modelo. Qual o nome do arquivo no qual são descritas as classes utilizadas para a geração das tabelas do banco de dados?**

A) urls.py.

B) views.py.

C) templates.py.

D) models.py.

E) settings.py.

4) Com o Django é possível configurar diferentes tipos de banco de dados para armazenar os dados das aplicações, inclusive com controle de múltiplos bancos simultaneamente. Marque a opção que representa a chave que deve receber como dicionário os parâmetros do banco de dados que será o banco padrão da aplicação.

A) ‘name’.

B) ‘engine’.

C) ‘default’.

D) ‘host’.

E) ‘port'.

5) O SGBD padrão do Django é o SQLite, que pode ser substituído pelo PostgreSQL ou pelo MySQL, entre outros bancos. Marque a opção que representa o valor que deve ser configurado na chave ‘ENGINE’ do arquivo settings.py para configurar o banco a fim de utilizar o MySQL.

A) 'db.backends.django.mysql'.

B) 'django.db.settings.mysql'.

C) 'db.django.backends.mysql'.

D) 'django.db.backends.oracle'.

E) 'django.db.backends.mysql'.

Na prática

É bastante comum aplicações serem desenvolvidas utilizando banco de dados para armazenar as informações apresentadas para os usuários e obtidas a partir dele por meio do preenchimento de formulários. Em alguns casos, existe a possibilidade da aplicação salvar os dados em mais do que um banco, o que pode ser facilmente resolvido, já que as configurações do Django permitem o cadastro de múltiplas bases de dados para serem utilizadas cada uma em determinada situação ou ambas ao mesmo tempo.

Neste Na Prática, você vai aprender como configurar o projeto criado com Django para trabalhar com o banco de dados MySQL no lugar do SQLite, banco padrão configurado inicialmente pelo Django.

ATUALIZANDO AS CONFIGURAÇÕES PARA USAR O BANCO MYSQL

Vinícius é um desenvolvedor sênior que, além de programação, trabalha com a criação inicial dos projetos de software para a sua equipe, bem como realiza as configurações para tirar o máximo de proveito das tecnologias de acordo com as necessidades das aplicações a serem desenvolvidas.

Certo dia, Vinícius percebeu em seus estudos um problema levantado pelo cliente: seria necessário **utilizar o MySQL como banco de dados e o *framework* Django poderia atender as suas necessidades caso a configuração inicial de banco fosse trocada para MySQL**.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Após configurar o ambiente de desenvolvimento com a instalação do MySQL e do client MySQL para o Django, Vinícius realizou a seguinte configuração para trabalhar com o MySQL:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'nome_do_banco_criado',
        'USER': '',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

Nesse caso, ele adicionou o motor do MySQL, configurou o nome do banco que já havia sido criado, bem como os dados de acesso ao banco.

Aponte a câmera para o código e acesse o link do conteúdo ou clique no código para acessar.