4 Flask e Banco de Dados

Resumo: *O tópico desta unidade é extremamente importante. Para a maioria dos aplicativos, haverá a necessidade de manter dados persistentes que possam ser recuperados eficientemente. E é exatamente por isso que os bancos de dados são aplicados.*

Flask em Banco de Dados

Como já advertimos anteriormente, `Flask` não oferece suporte nativo para banco de dados. Esta é uma das muitas áreas em que o `Flask` intencionalmente não opna, o que tem vantagens, pois temos liberdade de escolher o banco de dados que melhor se adapta ao aplicativo que pretendemos desenvolver, em vez de sermos forçados a nos adaptar a um específico.

Existem ótimas opções de banco de dados que funcionam em `Python` muitos deles com extensões `Flask` e que fazem uma excelente integração com o sistema. Os banco de dados podem ser separados em dois grandes grupos: aqueles que segem o modelo relacional e os que não seguem. O último grupo é frequentemente chamado de `NoSQL`, indicando que eles não implementam a popula linguagem de consulta relacional `SQL`. Embora existam ótimos produtos de banco de dados em ambos os grupos, ao longo das nossas aulas optei por utilizar o modelo de banco relacional por ser tal modelo uma combinação melhor para aplicativos que têm dados estruturados, como lista de usuários, postagens de blog, etc., enquanto os banco de dados `NoSQL` tendem a ser melhores para dados que apresentam uma estrutura menos definida. Porém, a depender do conhecimento do programador, qualquer tipo de banco de dados pode ser utilizado para o desenvolvimento de aplicativos.

Para a implementação de banco de dados na nossa aplicação precisaremos de duas novas extensões: o `Flask-SQLAlchemy` e o `Flask-Migrate`.

O `Flask-SQLAlchemy` fornece um *wrapper* amigável ao `Flask` para o pacote `SQLAlchemy`, que é um **Mapeador Relacional de Objetos** (ORM). Os ORMs permitem que os aplicativos gereciem um banco de dados usando entidades de alto nível, como classes, objetos e métodos, em vez de tabelas e `SQL`. O trabalho do ORM é traduzir as operações de alto nível em comandos de bancos de dados.

O legal do `SQLAlchemy` é que ele é um ORM não para um, mas para muitos bancos de dados relacionais. O `SQLAlquemy` suporta uma longa lista de mecanismos de banco de dados, incluindo os populares `MySQL`, `PostgreSQL` e `SQLite`. Isso é extremamente poderoso, porque você pode fazer seu desenvolvimento usando um banco de dados `SQLite` simples que não requer um servidor e, então, quando chegar a hora de implantar o aplicativo em um servidor de produção, podemos escolher um servidor `MySQL` ou `PostgreSQL` mais robusto, sem ter que alterar o aplicativo.

Para instalar o `Flask-SQLAlchemy` no nosso ambiente virtual, certifiqe-se de que o mesmo esteja ativado. Em seguida, execute o comando:

```
(venv) $ pip install flask-sqlalchemy
```

Migração de Banco de Dados

A maioria dos manuais de banco de dados focam na criação e uso dos bancos, e costumam deixar o problema das atualizações em um banco de dados em segundo plano. O problema é que conforme atualização vão sendo feitas ao longo do desenvolvimento do aplicativo as estruturas dos bancos são afetadas. Então, quando a estrutura muda, os dados que já estão no banco de dados precisam ser migrados para a estrutura nova.

Sendo assim a segunda extensão que iremos precisar a partir de agora é o `Flask-Migrate`. Essa extensão é um *wrapper* `Flask` para `Alembic`, uma estrutura de migração de banco de dados para o `SQLAlchemy`. Trabalhar com migrações de banco de dados adiciona um pouco de trabalho no início, mas esse é um pequeno preço a se pagar por uma composição mais robusta da nossa estrutura de dados ao fazer as alterações necessárias no nosso projeto de banco de dados.

O processo de instalação do `Flask-Migrate` é semelhante ao que já estamos familiarizados.

```
(venv) $ pip install flask-migrate
```

Configuração do Flask-SQLAlchemy

Durante o desenvolvimento, irei optar pelo uso do banco de dados `SQLite`, pois o mesmo é armazenado em um único arquivo e não há necessidade de executar um servidor de bando de dados como o `MySQL` ou o `PostgreSQL`.

O `Flask-SQLAlchemy` precisa de um novo item de configuração adicionado ao arquivo:

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))


class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'voce-nunca-saberah'
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'app.db')
```

A extensão `Flask-SQLAlchemy` pega a localização do bando de dados do aplicativo da variável de configuração `SQLALCHEMY_DATABASE_URI`. Como podemos lembrar da parte anterior, é em geral uma boa prática definiar a configuração a partir de variáveis de ambiente e fornecer um valor de *fallback*[1] quando o ambiente não define a variável. Neste caso, estamos pegando a URL do banco de dados da variável de ambiente `DATABASE_URL` e, se ela não estiver definida, configuramos um bando de dados chamado `app.db` localizado no diretório principal do aplicativo, que é armazenado na variável `basedir`.

[1] Una tradução livre do termo *fallback* seria **cair para trás**.

O banco de dados será representado no aplicativo pela instância do banco de dados. O mecanismo de migração do banco de dados também terá uma instância. Esses são objetos que precisam ser criados após o aplicativo, no arquivo `app/__init__.py`:

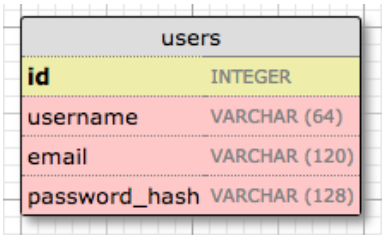
```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)

from app import routes, models
```

Note que fizemos três alterações no arquivo `app/__init__.py`. Primeiro, adicionamos um objeto `db` que representa o banco de dados. Depois, adicionamos `migrate`, para representar o mecanismo de migração do banco de dados. Observe ainda que existe um padrão em como devemos trabalhar com as extensões `Flask`. A maioria das extensões é inicizada como essas duas. Na última alteração, importamos um módulo novo chamado `models`. Este módulo irá definir a estrutura do banco de dados.

Modelos de Banco de Dados

Os dados que serão armazenados no banco de dados serão representados por uma coleção de classes, geralmente chamadas de modelos de banco de dados (*database models*). A camada **ORM** dentro do SQLAlchemy fará as traduções necessárias para mapear os objetos criados a partir dessas classes em linhas nas tabelas de banco de dados apropriadas.

Vamos começar criando um modelo que representa usuários. Usando a ferramenta SQL Designer, construimos o seguinte diagrama para representar os dados que queremos sar na tabela de usuários: O campo `id` geralmente está em todos os modelos e é usado como chave primária. Cada



usuário no banco de dados receberá um valor de `id` exclusivo, armazenado neste campo. As chaves primárias são, na maioria dos casos, atribuídas automaticamente pelo banco de dados, então só precisamos fornecer o campo `id` marcado como uma *chave primária*.

Os campos `username`, `email` e `password_hash` são definidos como strings (ou `VARCHAR` no jargão de banco de dados), e seus comprimentos máximos são especificados para que o banco de dados possa otimizar o uso do espaço. Embora os campos `username` e `email` sejam autoexplicativos, os campos `password_hash` merece alguma atenção. Precisamos ter certeza de que o aplicativo que

estamos construindo adota as melhores práticas de segurança e, por esse motivo, não armazenaremos as senhas de usuário em texto simples. O problema com o armazenamento de senhas é que, se o banco de dados for comprometido, os invasores terão acesso às senhas, e por isso pode ser devastador para os usuários. Em vez de escrever as senhas diretamente, vamos escrevr *hashes* de senhas, o que melhora muito a segurança. Este será o tópico de outro capítulo, então não se preocupe muito com isso por enquanto.

Agora que sabemos o que desejamos para a tabela de usuários, podemos traduzir isso em código no novo módulo `app/models.py`:

```
from typing import Optional
import sqlalchemy as sa
import sqlalchemy.orm as so
from app import db

class User(db.Model):
    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)
    email: so.Mapped[str] = so.mapped_column(sa.String(120), index=True,
                                             unique=True)
    password_hash: so.Mapped[Optional[str]] = so.mapped_column(sa.String(256))

    def __repr__(self):
        return '<User {}>'.format(self.username)
```

Começamos importando os módulos `sqlalchemy` e `sqlalchemy.orm` do pacote `SQLAchemy`, que fornecem a maioria dos elementos necessários para trabahar com um banco de dados. O módulo `sqlalchemy` inclui funções e classes de banco de dados de propósito geral, como tipos e auxiliares de construção de consulta, enquanto o múdulo `sqlalchemy.orm` fornece suporte para uso de modelos. Dado que esses dois nomes de módulo são longos e precisarão ser referenciados com frequência, utilizamos os *aliases* `sa` e `so` diretamente nas insturções de importação. A instância `db` do `Flask-SQLAlchemy` e a dica de digitação opcional do `Python` também são importadas.

A classe `User` criada acima representará os usuários armazenados no banco de dados. A classe herda de `db.Model`, uma classe base para todos os modelos do `Flask-SQLAlchemy`. O modelo `User` define vários campos como variáveis de classe. Essas são as colunas que serão criadas na tabela de banco de dados correspondente.

Os campos recebem um tipo uando dicas de tipo do `Python`, encapsuladas com o tipo genérico `so.Mapped` do `SQLAlchemy`. Uma declaração de tipo como `so.Mapped[int]` ou `so.Mapped[str]` define o tipo da coluna e também torna os valores obrigatórios ou não anuláveis em termos de banco de dados. para definir uma coluna que pode ser vazia ou anulável, o auxiliar `Optional` do `Python` também é adicionado, como `password_hash` demonstra.

Na maioria dos casos, definir uma coluna de tabela requer mais do que o tipo de coluna. O `SQLAlchemy` usa uma chamada de função `so.mapped_column()` atribuída a cada coluna para fornecer essa configuração adicional. No caso de `id` acima, a coluna é configurada como a **chave primária**. Para colunas de `string`, muitos bancos de dados exigem que um comprimento seja fornecido, então isso também está incluso. Incluímos outros argumentos opcionais que nos permitem indicar quais campos são exclusivos e indexados, o que é importante para que o banco de dados seja consistente e as pesquisas sejam eficientes.

O método `__repr__` diz ao `Python` como imprimir objetos dessa classe, o que será util para depuração. Podemos ver o método `__repr__()` em ação na sessão do interpretador `Python` abaixo:

```
>>> from app.models import User
>>> u = User(username='wanderson', email='wanderson@example.com')
>>> u
<User wanderson>
```

Criando o repositório de migração

A classe de modelo criada na seção anterior define a estrutura incial do banco de dados (ou esquema) par este aplicativo. Mas, à medida que o aplicativo continua a crescer é provável que precisemos fazer alterações nessa estrutura, com adicionar coisas novas e, às vezes, modificar ou remover itens. O `Alembic` (a estrutura de migração usada pelo `Flask-Migrate`) fará essas alterações de esquema de uma forma que não exija que o banco de dados seja recriado do zero toda vez que uma alteração for feita.

Para realiar essa tarefa aparentemente difícil, o `Alembic` mantém um repositório de migração, que é um diretório no qual ele armazena seus *scripts* de migração. Cada vez que uma alteração é feita no esquema de banco de dados, um *script* de migração é adicionado ao repositório com os detalhes da alteração. Para aplicar as migrações a um banco de dados, esses *scripts* de migração são executados na sequência em que foram criados.

O `Flask-Migrate` expõe seus comandos por meio do comando `flask`. Já vimos que o `flask run` é um subcomando nativo do `Flask`. O subcomando `flask db` é adicionado pelo `Flask-Migrate` para gerenciar tudo relacionado a migrações de bando de dados. Então, vamos criar o repositório de migração para o projeto, executando `flask db init`:

```
(venv) $ flask db init
  Creating directory /home/wanderson/uvv/migrations ... done
  Creating directory /home/wanderson/uvv/migrations/versions ... done
  Generating /home/wanderson/uvv/migrations/alembic.ini ... done
  Generating /home/wanderson/uvv/migrations/env.py ... done
  Generating /home/wanderson/uvv/migrations/README ... done
  Generating /home/wanderson/uvv/migrations/script.py.mako ... done
  Please edit configuration/connection/logging settings in
  '/home/wanderson/uvv/migrations/alembic.ini' before proceeding.
```

Lembrem-se de que o comando `Flask` depende da variável de ambiente `FLASK_APP` para saber onde o aplicativo `Flask` reside. Para este aplicativo, desejamos definir `FLASK_APP` para o valor `app.py`.

Depois de executar o comando `flask db init`, teremos um novo diretório chamado **migrations**, com alguns arquivos e um subdiretório **versions** dentro. Todos esses arquivos devem ser tratados como parte do nosso projeto a partir de agora e, em particular, devem ser adicionados ao controle de origem junto com o código do aplicativo.

A primeira migração

Com o repositório de migração em vigor, é hora de criar a primeira migração de banco de dados, que incluirá a tabela de usuários que mapeia para o modelo de banco de dados **Usuários**. Há

duas maneiras de criar uma migração de banco de dados: manualmente ou automaticamente. Para gerar uma migração automaticamente, o `Alembic` compara o esquema do banco de dados conforme definido pelos modelos de banco de dados, com o esquema de banco de dados real usado atualmente. Em seguida, ele preenche o *script* de migração com as alterações necessárias para fazer o esquema do banco de dados corresponder aos modelos do aplicativo. Nesse caso, como não há banco de dados anterior, a migração automática adicionará todo o modelo **Usuário** ao *script* de migração. O subcomando `flask db migrate` gera essas migrações automáticas:

```
(venv) $ flask db migrate -m "users table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'user'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_email' on '[↩
    'email']'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_username' on↩
     '['username']'
  Generating /home/wanderson/uvv/migrations/versions/e517276bb1c2_users_table.↩
      py ... done
```

A saída do comando dá uma ideia do que o `Alembic` incluiu na migração. As duas primeiras linhas são informativas e geralmente podem ser ignoradas. Em seguida, ele diz que encontrou uma tabela de usuário e dois índices. Novamente, a seguir ele nos informa onde escreveu o *script* de migração. O valor $e517276bb1c2$ é um código exclusivo e gerado automaticamente para a migração (será diferente para cada projeto ou cada execução). O comentári fornecido com a opção `-m` é opcional, ele apenas adiciona um pequeno texto descritivo à migração.

O *script* de migração gerado agora faz parte do projeto e, se você estiver usando o `git` ou outra ferramenta de controle de origem, ele precisa ser incorporado como um arquivo de origem adicional, junto com todos os outros arquivos armazenados no diretório `migrations`. Podemos ainda inspecinar o *script* para ver como ele se parece. Iremos descobrir que ele tem duas funções chamadas `upgrade()` e `downgrade()`. A função `upgrade()` aplica a migração e a função `downgrade()` a remove. Isso permite que o `Alembic` migre o banco de dados para qulquer ponto do histórico, mesmo para versões mais antigas, usando o caminho de `downgrade`.

O comando `flask db migrate` não faz nenhuma alteração no banco de dados, ele apenas gera o *script* de migração. Para aplicar as alterações no banco de dados, o comando `flask db upgrade` deve ser usado.

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> e517276bb1c2, users table
```

Como estamos utilizando o `SQLite` para as atividades de laboratório, o comando upgrade detectará que um banco de dados não existe e o criará (note que um arquivo chamado app.db será adicionado após a conclusão deste comando, que é o banco de dados `SQLite`). Ao trabalhar com servidores de banco de dados como *MySQL* e *PostgreSQL*, você precisa criar o banco de dados no servidor de banco de dados antes de executar o `upgrade`.

Observe que o `Flask-SQLAlchemy` usa uma convenção de nomenclatura *"snake case"* para tabelas de banco de dados padrão. Para o modelo **User** acima, a tabela correspondente no banco

de dados será chamada **user**. Para uma classe de modelo `AddressAndPhone`, a tabela seria chamada `address_and_phone`. Se você preferir escolher seus próprios nomes de tabela, pode adicionar um atributo chamado `__tablename__` à classe de modelo, definido como o nome desejado como uma *string*.

Fluxo de trabalho Upgrade e Downgrade do Banco de Dados

O aplicativo está em sua infância neste momento, mas não custa nada discutir qual será a estratégia de migração do banco de dados daqui para a frente. Imagine que temos o nosso aplicativo em uma máquina de desenvolvimento e também uma cópia implementada em um servidor de produção que está online e em uso.

Digamos que para a próxima versão do nosso aplicativo tenhamos que introduzir uma alteração em seus modelos, por exemplo, uma nova tabela precisa ser adicionada. Sem migrações, precisaríamos descobrir como alterar o esquema do banco de dados, tanto na máquina de desenvolvimento quanto no servidor de produção, e isso daria muito trabalho.

Porém, com o suporte à migração de banco de dados, depois de modificar os modelos no nosso aplicativo, podemos gerar um novo *script* de migração (`flask db migrate`), revisá-lo para ter certeza de que a geração automática fez a coisa certa e, em seguida, apliccar as alterações ao nosso banco de dados de desenvolvimento (`flask db upgrade`). Então, adicionaríamos o *script* de migração ao contro de oriem que o confirmará como válido.

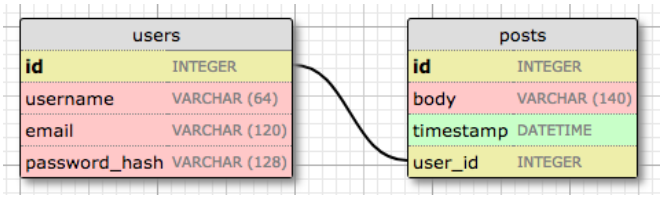
Conforme mencionamos, o comando `flask db downgrade` é capaz de desfazer a última migração. Embora seja improvável que precisemos dessa opção em um ambiente de produção, pode ser muito útil utilizá-la durante o desenovolvimento. Por exemplo, geramos um *script* de migração e o aplicamos, e em seguida descobrimos que as alterações feitas não são exatamente o que precisamos. Nesse caso, faremos o `downgrade` do banco de dados, excluiremos o *script* de migração e, em seguida, geramos um novo para substituí-lo.

Relacionamento de Banco de Dados

Banco de dados relacionais são bons para armazenar relações entre itens de dados. Considere o caso de um usuário escrevendo uma postagem de um *blog*. O usuário terá um registro na tabela `users` e a postagem terá um registro na tabela `posts`. A maneira mais eficiente de registrar quem escreveu uma determinada postagem é vincular os dois registros relacionados.

Uma vez que um *link* estre um usuário e uma postagem é estabelecido, o banco de dados pode resonder a consultas sobre esse *link*. O mais trivial é quando temos uma postagem de *blog* e precisamos saber qual usuário a escreveu. Uma consulta mais complexa é o inverso desta. Se temos um usuário, podemos desejar saber todas as postagens que esse usuário escreveu. O `SQLAlchemy` ajuda com ambos os tipos de consultas.

Vamos expandir o banco de dados para armazenar postagens de *blog* para ver os relacionamentos em ação. Aqui está o esquema para uma nova tabela `posts`:



A tabela `posts` terá o `id` necessário, o corpo do `post` e um `timespam`. Mas, além desses campos esperados, estamos adicionando um campo `user_id`, qe vincula o `post` ao seu autor. Vimos que todos os usuários têm uma **chave primária id**, que é única. A maneira de vincular um `post` de *blog* ao usuário que o criou é adicionar uma referência ao `id` do usuário, e é exatamente para isso que serve o campo `user_id`. Esse campo `user_id` é chamado de **chave estrangeira**, porque faz referência a uma chave primária de outra tabela. O banco de dados acima mostra chaves estrangeiras como um *link* entre o campo e o campo `id` da tabela à qual ele se refere. Esse tipo de relacionamento é chamado de **um para muitos**, por **um** usuário escreve **muitos** *posts*.

O `app/models.py` modificado é mostrado a seguir:

```
from datetime import datetime, timezone
from typing import Optional
import sqlalchemy as sa
import sqlalchemy.orm as so
from app import db

class User(db.Model):
    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)
    email: so.Mapped[str] = so.mapped_column(sa.String(120), index=True,
                                             unique=True)
    password_hash: so.Mapped[Optional[str]] = so.mapped_column(sa.String(256))

    posts: so.WriteOnlyMapped['Post'] = so.relationship(
        back_populates='author')

    def __repr__(self):
        return '<User {}>'.format(self.username)

class Post(db.Model):
    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    body: so.Mapped[str] = so.mapped_column(sa.String(140))
    timestamp: so.Mapped[datetime] = so.mapped_column(
        index=True, default=lambda: datetime.now(timezone.utc))
    user_id: so.Mapped[int] = so.mapped_column(sa.ForeignKey(User.id),
                                               index=True)

    author: so.Mapped[User] = so.relationship(back_populates='posts')

    def __repr__(self):
        return '<Post {}>'.format(self.body)
```

A nova classe `Post` representará postagens de *blog* escritas por usuários. O campo `timestamp` é definido com um tipo `datetime` e é configurado para ser indexado, o que é útil se você quiser recuperar postagens de forma eficiente em ordem cronológica. Também adicionamos um argumento padrão e passamos uma função `lambda` que retorna a hora atual no fuso horário UTC. Quando passamos uma função como padrão, o `SQLAlchemy` definirá o campo para o valor retornado pela função. Em geral, desejamos trabalhar com datas e horas UTC em um aplicativo de servidor em vez do horário local

onde estamos localizado. Isso nos garante um uso do `timestamps` uniformes, independentemente de onde os usuários e o servidor estejam localizados. Esses `timestamps` serão convertidos para o horário local do usuário quando forem exibidos.

O campo `user_id` foi inicializado como uma chave estrangeira para `User.id`, o que significa que ele faz referência a valores da coluna `id` na tabela `users`. Como nem todos os bancos de dados criam automaticamente um índice para chaves estrangeiras, a opção `index = True` é adicionada explicitamente, para que as pesquisas baseadas nesta coluna sejam otimizadas.

A classe `User` tem um novo campo `posts`, que é inicializado com `so.relationship()`. Este não é um campo de banco de dados real, mas uma visão de alto nível do relacionamento entre **usuários** e **posts**, e por esse motivo não está no diagrama do banco de dados. Da mesma forma, a classe `Post` tem um campo `author` que também é inicializado como um relacionamento. Esses dois atributos permitem que o aplicativo acesse as entradas de **usuário** e **post** conectadas.

O primeiro argumento para `so.relationship()` é a classe de modelo que representa o outro lado do relacionamento. Este argumento pode ser fornecido como uma *string*, o que é necessário quando a classe é definida posteriormente no módulo. Os argumentos `back_populates` referenciam o nome do atributo de relacionamento no outro lado, para que o `SQLAlchemy` saiba que esses atributos se referem aos dois lados do mesmo relacionamento.

O atributo de relacionamento `posts` usa uma **definição de tipagem diferente**. Em vez de `so.Mapped`, ele usa `so.WriteOnlyMapped`, que define `posts` como um tipo de coleção com objetos `Post`. Não se preocupe se esses detalhes não fizerem muito sentido ainda, mostrarei exemplos disso no final deste tema.

Como temos atualizações nos modelos de aplicativo, uma nova migração de banco de dados precisa ser gerada:

```
(venv) $ flask db migrate -m "posts table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'post'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_post_timestamp' ↩
    on
'['timestamp']'
  Generating /home/wanderson/uvv/migrations/versions/780739b227a7_posts_table.↩
     py ... done
```

E a migração precisa ser aplicada ao banco de dados:

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade e517276bb1c2 -> 780739b227a7,↩
    posts table
```

Brincando com o banco de dados

Até aqui fizemos um longo processo para definir o banco de dados, mas ainda não mostramos como tudo funciona. Como o aplicativo ainda não tem nenhuma lógica de banco de dados, vamos brincar com o banco de dados no interpretador `Python` para nos familiarizarmos com ele. Inicie o `Python`

no seu terminal (pode ser com o `iPython`). Certifique-se de que seu ambiente virtual esteja ativado antes de iniciar o interpretador.

Uma vez no prompt do `Python`, vamos importar o aplicativo, a instância do banco de dados, os modelos e o ponto de entrada do `SQLAlchemy`:

```
(venv) $ flask db migrate -m "posts table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'post'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_post_timestamp' ↩
    on
'['timestamp']'
  Generating /home/wanderson/uvv/migrations/versions/780739b227a7_posts_table.↩
     py ... done
```

E a migração precisa ser aplicada ao banco de dados:

```
>>> from app import app, db
>>> from app.models import User, Post
>>> import sqlalchemy as sa
```

O próximo passo pode parecer um pouco estranho. Para que o `Flask` e suas extensões tenham acesso ao aplicativo `Flask` sem ter que passar `app` como argumento em cada função, um contexto de aplicativo deve ser criado e enviado. Os contextos de aplicativo serão abordados em mais detalhes posteriormente, então, por enquanto, digite o seguinte código na sua sessão de *shell* do `Python`:

```
>>> app.app_context().push()
```

Em seguida, vamos criar um novo usuário:

```
>>> u = User(username='mario', email='mario@example.com')
>>> db.session.add(u)
>>> db.session.commit()
```

Alterações em um banco de dados são feitas no contexto de uma sessão de banco de dados, que pode ser acessada como `db.session`. Várias alterações podem ser acumuladas em uma sessão e, uma vez que todas as alterações tenham sido registradas, você pode emitir um único `db.session.commit()`, que grava todas as alterações atomicamente. Se a qualquer momento durante o trabalho em uma sessão houver um erro, uma chamada para `db.session.rollback()` abortará a sessão e removerá quaisquer alterações armazenadas nela. O importante a lembrar é que as alterações são gravadas no banco de dados somente quando um `commit` é emitido com `db.session.commit()`. As sessões garantem que o banco de dados nunca será deixado em um estado inconsistente.

Você está se perguntando como todas essas operações de banco de dados sabem qual banco de dados usar? O contexto do aplicativo que foi enviado acima permite que o `Flask-SQLAlchemy` acesse o aplicativo de instância do aplicativo `Flask` sem ter que recebê-lo como um argumento. A extensão procura no dicionário `app.config` pela entrada `SQLALCHEMY_DATABASE_URI`, que contém a URL do banco de dados.

Vamos adicionar outro usuário:

```
>>> u = User(username='joana', email='joana@example.com')
>>> db.session.add(u)
>>> db.session.commit()
```

O banco de dados pode responder a uma consulta que retorna todos os usuários:

```
>>> query = sa.select(User)
>>> users = db.session.scalars(query).all()
>>> users
[<User mario>, <User joana>]
```

A variável de consulta (`query`) neste exemplo recebe uma consulta básica que seleciona todos os usuários. Isso é obtido passando a classe de modelo para a função auxiliar de consulta `SQLAlchemy sa.select()`. Você verá que a maioria das consultas de banco de dados começa com uma chamada `sa.select()`.

A sessão de banco de dados, que acima foi usada para definir e confirmar alterações, também é usada para executar consultas. O método `db.session.scalars()` executa a consulta de banco de dados e retorna um iterador de resultados. Chamar o método `all()` do objeto de resultados converte os resultados em uma lista simples.

Em muitas situações, é mais eficiente usar o iterador de resultados em um *loop for* em vez de convertê-lo em uma lista:

```
>>> users = db.session.scalars(query)
>>> for u in users:
...     print(u.id, u.username)
...
1 mario
2 joana
```

Observe que os campos `id` foram automaticamente definidos como 1 e 2 quando esses usuários foram adicionados. Isso acontece porque o `SQLAlchemy` configura colunas de **chave primária** inteira para serem autoincrementadas.

Aqui está outra maneira de fazer consultas. Se você souber o `id` de um usuário, poderá recuperá-lo da seguinte maneira:

```
>>> u = db.session.get(User, 1)
>>> u
<User mario>
```

Agora vamos adicionar uma postagem de *blog*:

```
>>> u = db.session.get(User, 1)
>>> p = Post(body='meu primeiro post!', author=u)
>>> db.session.add(p)
>>> db.session.commit()
```

Não precisamos definir um valor para o campo `timestamp`, porque esse campo tem um padrão, que você pode ver na definição do modelo. E o campo `user_id`? Lembre-se de que o `so.relationship` que criamos na classe `Post` adiciona um atributo `author` às postagens. Atribuimos um `autor` a uma postagem usando esse campo `author` em vez de ter que lidar com IDs de usuário. O `SQLAlchemy` é ótimo nesse aspecto, pois fornece uma abstração de alto nível sobre relacionamentos e chaves estrangeiras.

Para concluir esta sessão, vamos dar uma olhada em mais algumas consultas de banco de dados:

```
>>> # Pegue todos os post escritos pelo usuario 1
>>> u = db.session.get(User, 1)
>>> u
<User mario>
>>> query = u.posts.select()
>>> posts = db.session.scalars(query).all()
>>> posts
[<Post meu primeiro post!>]

>>> # Vamos adicionar outros posts
>>> u = db.session.get(User, 2)
>>> u
<User joana>
>>> query = u.posts.select()
>>> posts = db.session.scalars(query).all()
>>> posts
[]

>>> #  Imprimir author posts e o corpo mensagem de todos
>>> query = sa.select(Post)
>>> posts = db.session.scalars(query)
>>> for p in posts:
...     print(p.id, p.author.username, p.body)
...
1 mario meu primeiro post!

# Pegue todos os usuarios em ordem alfabetica reversa
>>> query = sa.select(User).order_by(User.username.desc())
>>> db.session.scalars(query).all()
[<User mario>, <User joana>]

#  Pegue todos os usuarios que comecao com a letra "j"
>>> query = sa.select(User).where(User.username.like('j%'))
>>> db.session.scalars(query).all()
[<User joana>]
```

Observe como nos dois primeiros exemplos acima o relacionamento entre usuários e postagens é usado. Lembre-se de que o modelo `User` tem um atributo de relacionamento `posts` que foi configurado com o tipo genérico `WriteOnlyMapped`. Este é um tipo especial de relacionamento que adiciona um método `select()` que retorna uma consulta de banco de dados para os itens relacionados. A expressão `u.posts.select()` cuida da geração da consulta que vincula o usuário às suas postagens

de *blog*.

A última consulta demonstra como filtrar o conteúdo de uma tabela usando uma condição. A cláusula `where()` é usada para criar filtros que selecionam apenas um subconjunto das linhas da entidade selecionada. Neste exemplo, estamos usando o operador `like()` para selecionar usuários com base em um padrão.

A documentação do `SQLAlchemy` é o melhor lugar para aprender sobre as muitas opções disponíveis para consultar o banco de dados.

Para finalizar, saia do *shell* do `Python` e use os seguintes comandos para apagar os usuários de teste e postagens criados acima, para que o banco de dados esteja limpo e pronto para o próximo capítulo:

```
(venv) $ flask db downgrade base
(venv) $ flask db upgrade
```

O primeiro comando diz ao `Flask-Migrate` para aplicar as migrações do banco de dados na ordem inversa. Quando o comando `downgrade` não recebe um alvo, ele faz o `downgrade` de uma revisão. O alvo base faz com que todas as migrações sejam rebaixadas, até que o banco de dados seja deixado em seu estado inicial, sem tabelas.

O comando `upgrade` reaplica todas as migrações na ordem direta. O alvo padrão para atualizações é `head`, que é um atalho para a migração mais recente. Este comando efetivamente restaura as tabelas que foram rebaixadas acima. Como as migrações do banco de dados não preservam os dados armazenados no banco de dados, o `downgrade` e o `upgrade` têm o efeito de esvaziar rapidamente todas as tabelas.

Contexto do shell

Lembra do que no começo da seção anterior, logo após iniciar um interpretador `Python`, demos um `pool` para iniciar a instância do banco de dados? Tivemos que passar as seguintes linhas de comando para o interpretador:

```
>>> from app import app, db
>>> from app.models import User, Post
>>> import sqlalchemy as sa
>>> app.app_context().push()
```

Enquanto trabalhamos no nosso aplicativo, será preciso testar coisas em um *shell* `Python` com muita frequência, então ter que repetir as instruções acima toda vez vai ficar tedioso. Este é um bom momento para resolver esse problema.

O subcomando `flask shell` é outra ferramenta muito útil no guarda-chuva de comandos do `flask`. O comando *shell* é o segundo comando **"core"** implementado pelo `Flask`, depois do `run`. O propósito deste comando é iniciar um interpretador `Python` no contexto do aplicativo. O que isso significa? Veja o exemplo a seguir:

```
(venv) $ python
>>> app
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'app' is not defined
```

```
>>>

(venv) $ flask shell
>>> app
<Flask 'app'>
```

Com uma sessão de interpretador regular, o símbolo do aplicativo não é conhecido a menos que seja explicitamente importado, mas ao usar o `flask shell`, o comando pré-importa a instância do aplicativo e envia seu contexto de aplicativo para você. O legal do `flask shell` não é apenas que ele pré-importa o aplicativo, mas que você também pode configurar um **"contexto de shell"**, que é uma lista de outros símbolos para pré-importar.

A função a seguir em `app.py` cria um contexto de shell que adiciona a instância do banco de dados e os modelos à sessão de *shell*:

```
import sqlalchemy as sa
import sqlalchemy.orm as so
from app import app, db
from app.models import User, Post

@app.shell_context_processor
def make_shell_context():
    return {'sa': sa, 'so': so, 'db': db, 'User': User, 'Post': Post}
```

O decorador `app.shell_context_processor` registra a função como uma função de contexto de *shell*. Quando o comando `flask shell` é executado, ele invoca essa função e registra os itens retornados por ela na sessão de *shell*. O motivo pelo qual a função retorna um dicionário e não uma lista é que para cada item você também precisa fornecer um nome sob o qual ele será referenciado no *shell*, que é dado pelas chaves do dicionário.

Depois de adicionar a função do processador de **contexto de shell**, podemos trabalhar com entidades de banco de dados sem precisar importá-las:

```
(venv) $ flask shell
>>> db
<SQLAlchemy sqlite:////home/wanderson/uvv/app.db>
>>> User
<class 'app.models.User'>
>>> Post
<class 'app.models.Post'>
```

Se você tentar executar o código acima e receber exceções `NameError` ao tentar acessar `sa`, `so`, `db`, `User` e `Post`, então a função `make_shell_context()` não está sendo registrada com o `Flask`. A causa mais provável disso é que você não definiu `FLASK_APP=app.py` no ambiente.